CS240 Programming in C

Gustavo Rodriguez-Rivera
Purdue University

EMERGENCY PREPAREDNESS – A MESSAGE FROM PURDUE

To report an emergency, call 911. To obtain updates regarding an ongoing emergency, sign up for Purdue Alert text messages, view www.purdue.edu/ea.

There are nearly 300 Emergency Telephones outdoors across campus and in parking garages that connect directly to the PUPD. If you feel threatened or need help, push the button and you will be connected immediately.

If we hear a fire alarm during class we will immediately suspend class, evacuate the building, and proceed outdoors. Do not use the elevator.

If we are notified during class of a Shelter in Place requirement for a tornado warning, we will suspend class and shelter in [the basement].

If we are notified during class of a Shelter in Place requirement for a hazardous materials release, or a civil disturbance, including a shooting or other use of weapons, we will suspend class and shelter in the classroom, shutting the door and turning off the lights.

Please review the Emergency Preparedness website for additional information.
http://www.purdue.edu/ehps/emergency_preparedness/index.html

General Information

- Web Page: http://www.cs.purdue.edu/homes/cs240
- Office: LWSN1210
- E-mail: grr@cs.purdue.edu
- Textbook:
  - “The C programming Language” (Second Edition) by Kernighan and Ritchie.

Mailing List

- All announcements will be sent via email and posted in Piazza.
- Mailing List will be created automatically

Labs

- Lab 1 is already posted. It is due Monday.
- Post your questions in Piazza
- You may attend the second hour of any lab to ask for help.
- Also you may attend office hours of GTAs and UTAs.
- Office hours are posted in the webpage.

Lab Policy

- Do your project on your own.
- It is OK to discuss about your labs but at the time of coding do it on your own.
- We will check projects for academic dishonesty.
- Academic Dishonest will be penalized with failing the course and reporting to the Dean of Students.

Grading

- Grade allocation
  - Midterm: 25%
  - Final: 25%
  - Projects: 50%
- Exams will include programming questions.

Course Contents

- C development cycle:
- C Programming Structure
- Control Flow
- While/for/do, etc
- Functions
- Arrays & Strings
- Pointer
- Dynamic memory
- Data Types: structures, unions, strings
- Streams (Files)
- File calls/ directories
- UNIX Intro makefiles
- Processes & Threads
- Sockets
- C++

The C Language

- C was created by Dennis Ritchie in 1972 in Bell Labs
- C was used to implement UNIX
- Operating Systems used to be implemented 100% in assembly language making them architecture dependent.
- C was designed to make UNIX portable:
  - 95% of Unix is in C
  - 5% is in assembly
- Only the assembly language part needs to be rewritten to migrate to other machine.
- Most of the optimizations you can do in assembly language you can do them in C.
- C is a “High-Level” assembly language.

Uses of C

- Linux is written in C
- Most of the libraries (low level) that need speed are written in C, Graphics (OpenGL), MP3 decoders, math libraries are also written in C
- Java runs on top of the JVM and the JVM is written in C
- Games that need high performance are written in C (or C++ in some cases or Objective-C)
- Maybe the only popular game written in Java is Minecraft. All other ones are in C/C++/Objective C.
- Java syntax was based on C&C++, therefore, you will find that many of the elements in C you knew them already
- C++ is a superset of C, so by learning C, you learn a big chunk of C++.
- C is used in Java native libraries that need speed. E.g. User Interface, Access to Database, Animation, Rendering, Math Library etc.

The C Principle

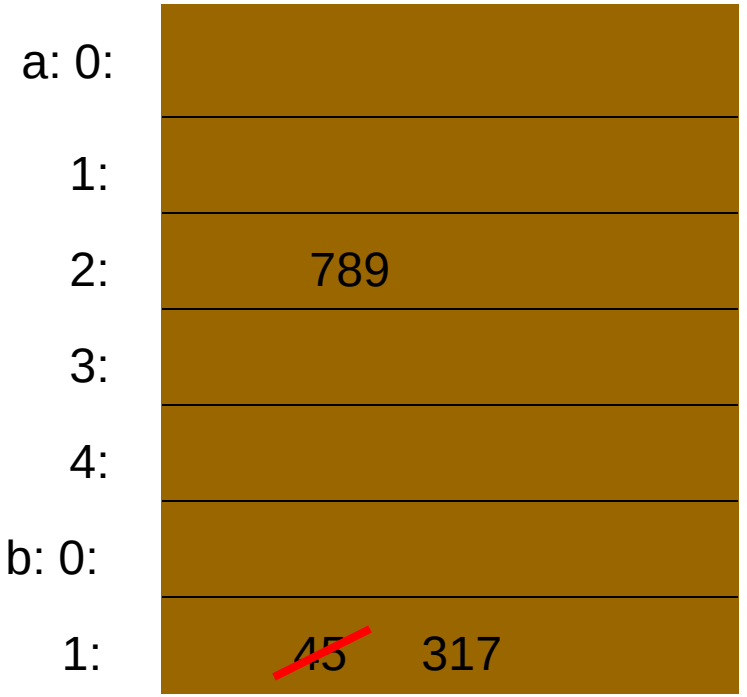
"C will not get in your way to make your program run fast."

- For example an array access such as :
  a[i]=s;
- In Java
  // Check boundaries
  if (i >= 0 && i<max) { a[i]=s; } else throw out of boundary exception
- In C
  no checks!!!!
  Assignment will take place in memory even if i is out of range. Assignment will happen beyond the end of the array.
- An out-of-bounds is very difficult to debug since the assignment may happen in different variable located in memory just after the array.
- The same advantage of C that makes it fast makes it vulnerable to safety problems.

Out-of-bounds assignment in C

```
int a[5];
int b[3];
a[2]=789;
b[1]=45;
a[6]=317;
```

a and b in memory



b[1] will be overwritten !!!!!

C principle revisited

"C will not get in your way to make your program run fast."

…. However, C will not protect you if you make a mistake!!!

- You have to know what you are doing.
- Programming and debugging in C is more difficult and time consuming than programming in Java
- Java is used in applications that do not require too much CPU (I/O bound). Example: Web apps, calendar app
- C is used in applications that require a lot of CPU (CPU bound). Example: Games, MP3 Player.

Memory Usage in C and Java

- Java uses Garbage Collection.
  - The JVM will collect the objects that are unreachable by the program.
- C uses explicit memory management.
  - After calling p=malloc(size) to allocate memory, you will have to call free(p) when object p is no longer in use or it will continue using memory in the program.
  - This is called a memory leak.
  - If the program has a lot of memory leaked, the execution will slow down due to excessive memory usage or even crash.
  - If your program calls free(p) and you write to the object pointed by p then the memory allocator data structures may get corrupted and your program will crash.
  - This is called a premature free.

- ***Memory allocation errors make programming in C difficult.***

Memory Usage in C and Java

- C programs in general use less that half the size of a Java program.
- C programs can be “Fast and Lean” but you have to be careful writing them.
- In general when having a new project, try to write it in Java, C#, Python etc.
- Only if speed is required use C.

Example of a C Program

- Use a text editor to create the file and name it hello.c

```
#include <stdio.h>  //include file from
/usr/includes/stdio.h
int main()
{
    printf("Hello world\n");
}
```

- To edit file use gedit, pico, vim, or xemacs.
- Pico and vim can be used in a text terminal using ssh from home.
- Gedit and xemacs need a windows system so they only can be used in the lab machines or running Linux at home.

Compiling a C program

- To compile a program

  gcc -o hello hello.c

  “hello” is the name of the executable.
- Also you may use

  gcc –g –o hello hello.c

  - Compile with debug information

  gcc -std=gnu99 –o hello hello.c

  - Compiles against the GNU C99 standard,
- To run it:

  bash% ./hello

  Hello World

Example: min/max

```
#include <stdio.h>

main() {
        printf("Program that prints max and min of two numbers
   a,b\n");
        int a, b;
        int max,min;

        while (1) {
                printf("Type a and <enter>: ");
                scanf("%d",&a);
                getchar(); // Discard new line

                printf("Type b and <enter>: ");
                scanf("%d",&b);
                getchar(); // Discard new line
```

Example: min/max (cont.)

```
        if (a > b) {
                max = a;
                min = b;
        }
        else {
                max = b;
                min = a;
        }
        printf("max=%d min=%d\n",max,min);

        printf("Do you want to continue? Type y/n and <enter>");
        char answer;
        answer = getchar();

        if (answer=='n') {
                break;
        }
    }
    printf("Bye\n");
}
```

Example: min/max (cont.)

```
cs240@data ~/2014Fall/lab1/lab1-src $ gcc -o minmax  minmax.c
cs240@data ~/2014Fall/lab1/lab1-src $ ./minmax
Program that prints max and min of two numbers a,b
Type a and <enter>: 7
Type b and <enter>: 3
max=7 min=3
Do you want to continue? Type y/n and <enter>y
Type a and <enter>: 9
Type b and <enter>: 5
max=9 min=5
Do you want to continue? Type y/n and <enter>n
Bye
```

Example: Implementing "grep"

- ***grep*** is a UNIX command that is used to print the lines of a file that match a given pattern.

  grep pattern file

- Example:

```
lore 141 $ grep size index.html
  <frame name="left" scrolling="no" noresize
target="rtop" src="c.htm">
  <frame name="rtop" scrolling="no" target="rbottom"
src=".htm" noresize>
```

Mygrep implementation

```
/*
 * mygrep: Print the lines of a file that match a string
 *
 * mygrep pattern file
 */
#include <stdio.h>
#define MAXLINE 1023 //define a marco
char line[MAXLINE+1]; //global variable

void mygrep(char * fileName, char * pattern); //forward definition (prototype)

int main(int argc, char **argv)
{
    char * fileName;
    char * pattern;
    // Check that there are at least 2 arguments
    // mygrep file pattern
    // argv[0] argv[1] argv[2]
    // argc == 3
    if (argc<3) {
        printf("Usage: mygrep pattern file\n");
        exit(1);
    }
    pattern = argv[1];
    fileName = argv[2];
    mygrep(fileName, pattern);
    exit(0);
}
```

Mygrep implementation (cont.)

```
void mygrep(char * fileName, char * pattern) {
    FILE * fd = fopen(fileName,"r");
    if (fd == NULL) {
        printf("Cannot open file %s\n", fileName);
        exit(1);
    }
    while(fgets(line, MAXLINE, fd)!= NULL) {
        if (strstr(line,pattern)!=NULL) {
            printf("%s", line);
        }
    }
    fclose(fd);
}
```

Bits and Bytes

- Bits are grouped in bytes, where each byte is made of 8 bits.
- In modern computers a byte is the smallest unit that can be addressed by a CPU.
- A byte can be used to store values such as 00000000 (0 in decimal) to 11111111 (255 in decimal).
- These are very small numbers, so usually larger groups of bytes are used to represent other types of data .

Representation of Numbers in Memory

- Integers are represented in groups of
  - 1 byte (char)
  - 2 bytes (short int or short),
  - 4 bytes (int) ,
  - 8 bytes (long int or long)
  - and in some architectures 16 bytes (long long int or long long) variables.
  - Example of an 8 byte number in memory

```
00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
63       55       47       39       31       23       15       7       0
```

$2^{23}+2^{19}+2^{16}+2^{13}+2^{10}+2^{7}+2^{4}+2^{1}= 8987794$

Representation of Negative Numbers in Memory

- Negative numbers typically use a representation called “complements of two”,
- A negative number is obtained by inverting the corresponding positive number and then adding 1.
- This representation allows using common positive integer arithmetic to do the addition and subtraction operations.
- For instance, the number represented above as a negative number can be obtained as:

Original: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
Negated: 11111111 11111111 11111111 11111111 11111111 01110110 11011011 01101101
Plus 1 11111111 11111111 11111111 11111111 11111111 10001001 00100100 10010010
63 55 47 39 31 23 15 7 0

$$2^{23}+2^{19}+2^{16}+2^{13}+2^{10}+2^{7}+2^{4}+2^{1}=$$
-8987794

Complements of Two and Addition

- If we have the binary representation of 8987794 ***added*** to same number in complements of two representing -8987794 we will obtain 0 as expected.

```
 8987794: 00000000 00000000 00000000 00000000 00000000 10001001 00100100 10010010
+
-8987794: 11111111 11111111 11111111 11111111 11111111 10001001 00100100 10010010
 -------  --------------------------------------------------------------------------
       0  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

Representation of Strings

- Basic strings in C language are represented in memory as a sequence of bytes delimited by a 0 value.
- Each byte represents a character in ASCII representation.
- ASCII is the standard that translates characters in the English alphabet to numbers. ASCII stands for American Standard Code for Information Interchange.

ASCII Table

32:	48:0	64:@	80:P	96:`	112:p
33:!	49:1	65:A	81:Q	97:a	113:q
34:"	50:2	66:B	82:R	98:b	114:r
35:#	51:3	67:C	83:S	99:c	115:s
36:$	52:4	68:D	84:T	100:d	116:t
37:%	53:5	69:E	85:U	101:e	117:u
38:&	54:6	70:F	86:V	102:f	118:v
39:'	55:7	71:G	87:W	103:g	119:w
40:(	56:8	72:H	88:X	104:h	120:x
41:)	57:9	73:I	89:Y	105:i	121:y
42:*	58::	74:J	90:Z	106:j	122:z
43:+	59:;	75:K	91:[	107:k	123:{
44:,	60:<	76:L	92:\	108:l	124:|
45:-	61:=	77:M	93:]	109:m	125:}
46:.	62:>	78:N	94:^	110:n	126:~
47:/	63:?	79:O	95:_	111:o	127:

C String Representation

- For example, the string “Hello world is represented by the equivalent ASCII characters delimited by a NULL character.

	H	e	l	l	o	\0
Bytes	72	101	108	108	111	0
Addr:	800	801	802	803	804	805

UNICODE

- To be able to represent characters in other languages, the Unicode standard was created.
- Unicode extends the ASCII standard and it uses two bytes to represent a character instead of one.
- In Unicode it is possible to represent the characters of mostly all languages in the world.

Memory of a Program

- From the point of view of a program, the memory in the computer is an array of bytes
- This array goes from address 0 to address 2^{64}-1 (0x0 to 0xFFFFFFFFFFFFFFFF) assuming a 64-bit architecture.

Computer Memory as an Array of Bytes

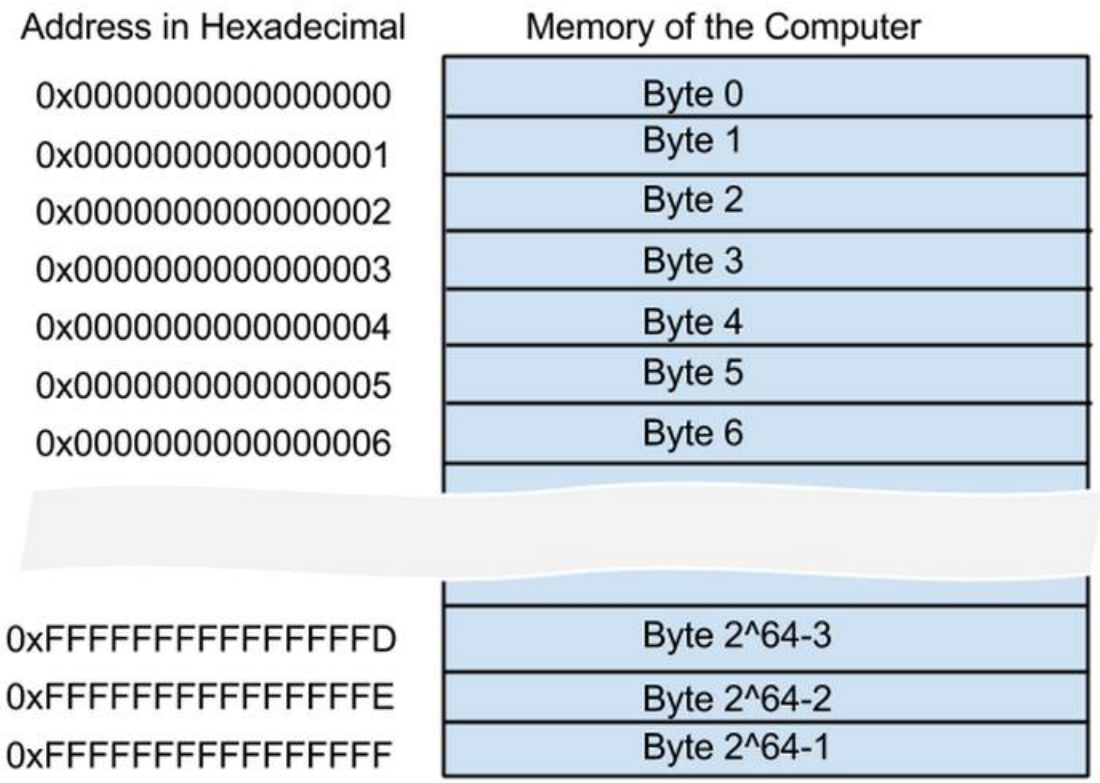


Figure 1.1: Computer Memory as an Array of Bytes (char[])

Memory Gaps

- Every program that runs in memory will see the memory this way.
- In C/C++ or assembly language it is possible to access the location of any of these bytes using pointers and pointer dereferencing.
- Theoretically a program may access any of these locations.
- However, there are gaps in the address space.
- Not all the addresses are “mapped” to physical memory.
- When accessing memory in these gaps, the program will get an exception called Segmentation Violation or SEGV and the program will crash.

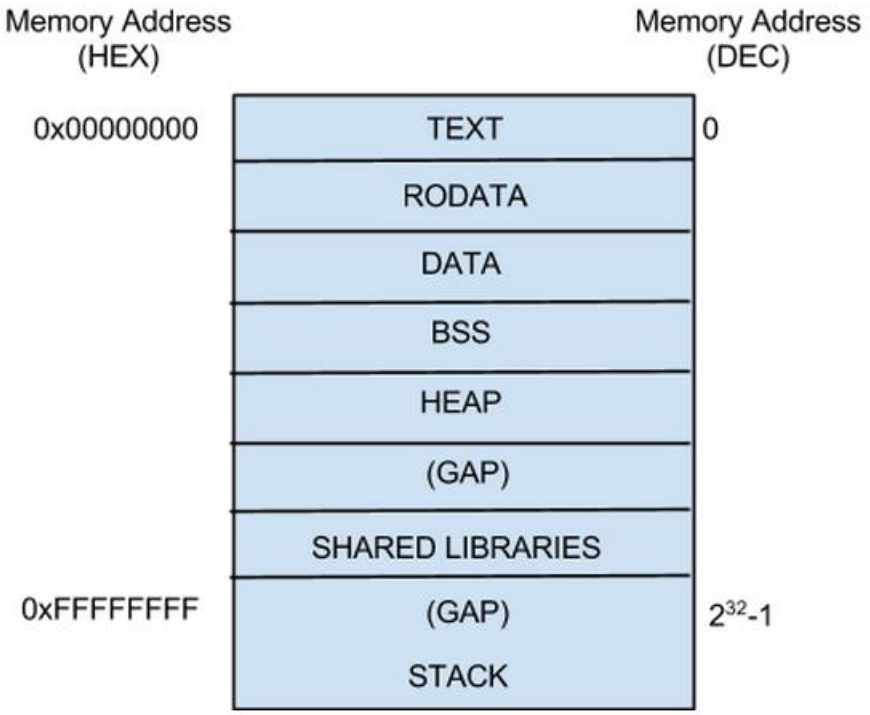
Data Types and Bytes

- Types such as integers, floats, doubles, or strings are represented as one or more of these bytes in memory.
- Everything stored in memory is represented with bytes.
- In C it is up to the program and the programmer to give meaning to what is stored in these bytes in memory.

Sections of a Program

- The memory of the computer is used to store both the program code, and the data that the program manipulates.
- An executable program in memory is divided into sections.
  - TEXT - Instructions that the program runs
  - RODATA - Stores Read-only data. These are constants that the program uses, like strings constants or other variables defined like “const int
  - DATA – Initialized global variables.
  - BSS – Uninitialized global variables. They are initialized to zeroes.
  - HEAP – Memory returned when calling malloc/new. It grows upwards.
  - SHARED LIBRARIES – Also called dynamic libraries. They are libraries shared with other processes.
  - STACK – It stores local variables and return addresses. It grows downwards.

Sections of a Program



Address Space

- Each program has its own view of the memory that is independent of each other.
- This view is called the “Address Space” of the program.
- If a process modifies a byte in its own address space, it will not modify the same location of the address space of another process.

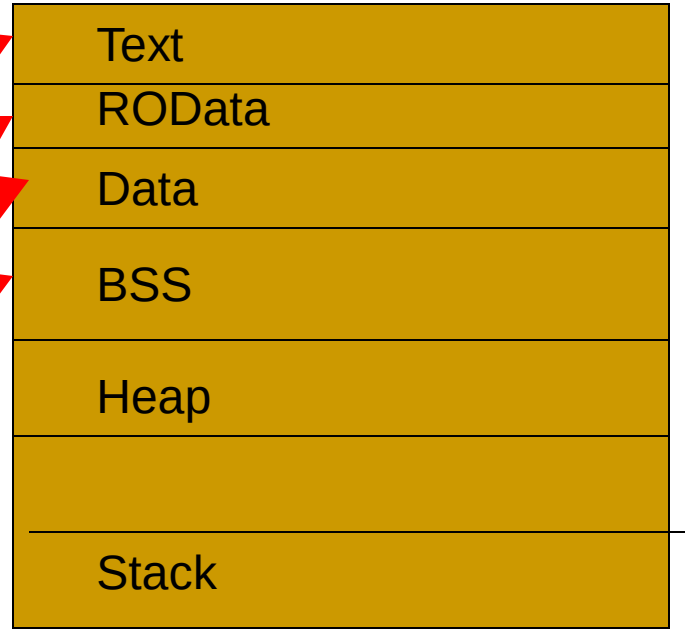
Printing Program Memory Addresses

```
#include <stdlib.h>
#include <stdio.h>

int a = 5; // Stored in data section
int b[20]; // Stored in bss
const char * hello = "Hello world";

int main() { // Stored in text
        int x; // Stored in stack
        int *p =(int*) malloc(sizeof(int)); //Stored in heap
        printf("sizeof(int)=%ld\n", sizeof(int));
        printf("sizeof(long)=%ld\n", sizeof(long));
        printf("sizeof(int*)=%ld\n", sizeof(int*));
        printf("(TEXT) main=0x%lx\n", (unsigned long)main);
        printf("(ROData) Hello=0x%lx\n", (unsigned long)hello);
        printf("(Data) &a=0x%lx\n", (unsigned long)&a);
        printf("(Bss) &b[0]=0x%lx\n", (unsigned long)&b[0]);
        printf("(Heap) p=0x%lx\n", (unsigned long)p);
        printf("(Stack) &x=0x%lx\n", (unsigned long)&x);
}
```

Note: &x means the address of variable x or where x is stored in memory.

Printing Program Memory Addresses

```
cs240@data ~/2014Fall/LectureNotes/test $ ./test1
sizeof(int)=4
sizeof(long)=8
sizeof(int*)=8
(TEXT) main=0x4005bc
(RODataa) Hello=0x400744
(Data) &a=0x601048
(Bss) &b[0]=0x601080
(Heap) p=0x1075010
(Stack) &x=0x7fff93cdd4d4
```



Simple Memory Dump of a Program

- In Lab2 you will write a memory dump function that will print the memory dump of a program:

Hint: Use char *p like it was an array

hintdump.c:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void mymemdump(FILE * fd, char * p , int len) {
   int i;
   fprintf(fd, "0x%016lX: ", (unsigned long) p); // Print address of the beginning of p. You need to print it every 16 bytes
   for (i=0; i < len; i++) {
      int c = p[i]&0xFF; // Get value at [p]. The &0xFF is to make sure you truncate to 8bits or one byte.

      // Print first byte as hexadecimal
      fprintf(fd, "%02X ", c);

      // Print first byte as character. Only print characters >= 32 that are the printable characters.
      fprintf(fd, "%c ", (c>=32)?c:'.');

     if (i % 16 == 0 ) {
      fprintf(fd,"\n");
    }
   }
}

main() {
   char a[30];
   int x;
   x = 5;
   strcpy(a,"Hello world\n");
   mymemdump(stdout,(char*) &x, 64);
}
```

Simple Memory Dump of a Program

```
cs240@data ~/2014Fall/lab2 $ gcc -o hintdump hintdump.c
cs240@data ~/2014Fall/lab2 $ ./hintdump
0x00007FFF150B1A9C: 05 .
00 . 00 . 00 . 48 H 65 e 6C l 6C l 6F o 20   77 w 6F o 72 r 6C l 64 d 0A . 00 .
00 . 00 . 00 . A0  1B . 0B . 15 . FF  7F  00 . 00 . 00 . 00 . 00 . 00 . 00 .
00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . A5  3C < 35 5 49 I 63 c
7F  00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . 00 . A8  1B . 0B . 15 .
cs240@data ~/2014Fall/lab2 $

The challenge for lab2 is to make the output of mymemdump look like this:

cs240@data ~/2014Fall/lab2/lab2-src $ ./mem
&x=0x7FFF89A62890
&y=0x7FFF89A628A8
0x00007FFF89A62890: 41 33 40 50 09 00 00 00 30 06 9C 50 D7 7F 00 00  A3@P....0.P..
0x00007FFF89A628A0: 94 28 A6 89 FF 7F 00 00 00 00 00 00 00 00 28 40  (........(@
0x00007FFF89A628B0: 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00  Hello world.....
0x00007FFF89A628C0: FF B2 F0 00 00 00 00 00 00 00 00 00 00 00 00 00  ..............
head=0x1e83010
```

Examining the memory of the program:

```
cs240@data ~/2014Fall/lab2/lab2-src-sol $ cat mem.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void mymemdump(FILE *fd, char * p , int len);

struct X{
  char a;
  int i;
  char b;
  int *p;
};

struct List {
  char * str;
  struct List * next;
};

int
main() {
  char str[20];
  int a = 5;
  int b = -5;
  double y = 12;
  struct X x;
  struct List * head;
```

Examining the memory of the program:

```
  x.a = 'A';
  x.i = 9;
  x.b = '0';
  x.p = &x.i;
  strcpy(str, "Hello world\n");
  printf("&str=0x%lX\n", (unsigned long)str);
  printf("&a=0x%lX\n", (unsigned long)&a);
  printf("&b=0x%lX\n", (unsigned long)&b);
  printf("&x=0x%lX\n", (unsigned long)&x.a);
  printf("&y=0x%lX\n", (unsigned long) &y);

  mymemdump(stdout, (char *) &x.a, 64);
  head = (struct List *) malloc(sizeof(struct List));
  head->str=strdup("Welcome ");
  head->next = (struct List *) malloc(sizeof(struct List));
  head->next->str = strdup("to ");
  head->next->next = (struct List *) malloc(sizeof(struct List));
  head->next->next->str = strdup("cs250");
  head->next->next->next = NULL;
  printf("head=0x%lx\n", (unsigned long) head);
  mymemdump(stdout, (char*) head, 128);
}
```

Examining the memory of the program

```
cs240@data ~/2014Fall/lab2/lab2-src-sol $ ./mem
&str=0x7FFFCFB29B50
&a=0x7FFFCFB29B4C
&b=0x7FFFCFB29B48
&x=0x7FFFCFB29B20
&y=0x7FFFCFB29B40
0x00007FFFCFB29B20: 41 E3 D1 41 09 00 00 00 30 B6 2D 42 30 7F 00 00  AA....0-B0..
0x00007FFFCFB29B30: 24 9B B2 CF FF 7F 00 00 B7 B1 DA 41 30 7F 00 00  $..A0..
0x00007FFFCFB29B40: 00 00 00 00 00 00 28 40 FB FF FF FF 05 00 00 00  ......(@....
0x00007FFFCFB29B50: 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 00 00 00  Hello world.....
head=0x1adc010
0x0000000001ADC010: 30 C0 AD 01 00 00 00 00 50 C0 AD 01 00 00 00 00  0.....P.....
0x0000000001ADC020: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC030: 57 65 6C 63 6F 6D 65 20 00 00 00 00 00 00 00 00  Welcome ........
0x0000000001ADC040: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC050: 70 C0 AD 01 00 00 00 00 90 C0 AD 01 00 00 00 00  p...........
0x0000000001ADC060: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
0x0000000001ADC070: 74 6F 20 00 00 00 00 00 00 00 00 00 00 00 00 00  to ..............
0x0000000001ADC080: 00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00  ........!.......
cs240@data ~/2014Fall/lab2/lab2-src-sol $

Colors:
   str a b

Variable            Address         Value
------------------- --------------- -------------
str                 0x7FFFCFB29B50 48 65 6C 6C 6F 20 77 6F 72 6C 64 0A 00 Hello world
a                   0x7FFFCFB29B4C 05 00 00 00  (5)
b                   0x7FFFCFB29B48 FB FF FF FF  (-5)
```

Notes on Lab2

Important for lab2: Copy the new mem.c from

https://www.cs.purdue.edu/homes/cs240/lab2/lab2-src/mem.c

The new one mem.c prints the addresses of str, a, b. If you don't do that, the compiler will eliminate the unused variables from the executable and they will not show in memory.

You may modify mem.c to print the addresses of other elements such as head , head->str head->next head->next->str etc.

What is GDB

- GDB is a debugger that helps you debug your program.
- The time you spend now learning gdb will save you days of debugging time.
- A debugger will make a good programmer a better programmer.

Compiling a program for gdb

- You need to compile with the “-g” option to be able to debug a program with gdb.
- The “-g” option adds debugging information to your program

```
gcc –g –o hello hello.c
```

Running a Program with gdb

- To run a program with gdb type

  `gdb progname`

  `(gdb)`
- Then set a breakpoint in the main function.

  `(gdb) break main`
- A breakpoint is a marker in your program that will make the program stop and return control back to gdb.
- Now run your program.

  (**`gdb) run`**
- If your program has arguments, you can pass them after run.

Stepping Through your Program

- Your program will start running and when it reaches “main()” it will stop.

```
gdb>
```

- Now you have the following commands to run your program step by step:

(gdb) step

It will run the next line of code and stop. If it is a function call, it will enter into it

(gdb) next

It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Example:

```
(gdb) step
(gdb) next
```

Setting breakpoints

- You can set breakpoints in a program in multiple ways:

  (gdb) break function

  Set a breakpoint in a function E.g.

  (gdb) break main

  (gdb) break line

  Set a break point at a line in the current file. E.g.

  (gdb) break 66

  It will set a break point in line 66 of the current file.

  (gdb) break file:line

  It will set a break point at a line in a specific file. E.g.

  (gdb) break hello.c:78

Regaining the Control

- When you type

  `(gdb) run`

  the program will start running and it will stop at a break point.

- If the program is running without stopping, you can regain control again typing ctrl-c.

Where is your Program

- The command

  (gdb)where

  Will print the current function being executed and the chain of functions that are calling that fuction.

  This is also called the backtrace.

  Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Printing the Value of a Variable

- The command

  (gdb) print var

  Prints the value of a variable.

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Exiting gdb

- The command “quit” exits gdb.

```
(gdb) quit
The program is running.  Exit
  anyway? (y or n) y
```

Debugging a Crashed Program

- This is also called “postmortem debugging”
- It has nothing to do with CSI ☺
- When a program crashes, it writes a **core file**.

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

- The core is a file that contains a snapshot of the program at the time of the crash. That includes what function the program was running.

Debugging a Crashed Program

- Sometimes the sysadmins disable the generation of core files to reduce the disk space waste. This happens in the CS machines.
- To find out if your system is able to generate cores type:

```
grr@data ~/cs240 $ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
```

- If you see that the core file size is 0. Enable core file generation by typing:

  grr@data ~/cs240 $ ulimit -c 1000000

Debugging a Crashed Program

- To run gdb in a crashed program type

  `gdb program core`

  E.g.

```
bash-4.1$ gdb hello core
GNU gdb 6.6
Program terminated with signal 11, Segmentation fault.
#0  0x000106cc in main () at hello.c:11
11          *s2 = 9;
(gdb)
```

  - Now you can type ***where*** to find out where the program crashed and the value of the variables at the time of the crash.

```
(gdb) where
#0  0x000106cc in main () at hello.c:11
(gdb) print s2
$1 = 0x0
(gdb)
```

  - This tells you why your program crashed. Isn't that great?

Now Try gdb in Your Own Program

- Make sure that your program is compiled with the –g option.
- Remember:
  - One hour you spend learning gdb will save you days of debugging.
  - Faster development, less stress, better results

Structure of a C program

- A C program is a sequence of definitions of functions, and global variables, type definitions and function prototypes:

  global var1
  global var2
  func1
  func2
  func3

- Example:

```
int sum; // Global variable. Lifetime is the entire duration of the program.
int sum(int a, int b) {
  int x; // Local variable. Lifetime is only when this funciton is invoked.
  x = a + b;
  return x;
}
int main() {
{
    int y;
    y = sum(3,4)
   printf("y=%d\n", y);
}
```

- C does not have classes interfaces etc.

Compiling from multiple files

- Compiling object files:
- You may compile each file separately and then link them:

```
gcc –c sum.c          - Generates sum.o
gcc –c main.c         - Generates main.o
gcc –o main main.o main.c  - Generates
  main
```

- Alternatively you may generates main in a single step.

```
gcc –o main main.c sum.c
```

- The first option is preferred if many .o files are used and only a few are modified at a time.

Functions in Multiple Files

- A function defined in one file can be used in another file.

sum.c:

```
extern int total; // extern declaration
void sum( int a, int b) {
   total = a + b;
}
```

- The second file needs to have an extern definition:

main.c:

```
#include <stdio.h>
int sum; //definiton
extern sum(int a, int b);
main() {
  printf("sum=%d\n", sum(5,8));
}
```

Making functions private

- You can make a function private to a file by adding the *static* keyword before the function declaration.

```
static void mylocalfunction(int x) {
  ….
}
// The mylocalfunction() function can only be used in
   this file.
```

- In this way no other C file can see this function.

Scope and Lifetime of a variable

- Scope = The *place* in the program where a variable can be used.
- Lifetime = The *time* in the execution when a variable is valid.

Global Variables

- Global Variables are defined outside a function.
- *The Scope* of a Global Variable is from where it is defined to the end of the file.
- *The Lifetime* of a Global Variable is the whole duration of the program.
- If the C program spans more than two files a global variable can be used in both files
- One file has the *definition*:

```
int total;
```

- The other file may have an “extern” declaration to tell that the variable is defined in another file.

```
extern int total;
```

- The extern declaration in this case is optional but recommended.
- Global Variables are intialized with 0s.

Global Variables in Multiple Files

sum.c:

```
extern int total; // extern declaration
void sum( int a, int b) {
   total = a + b;
}
```

---

main.c:

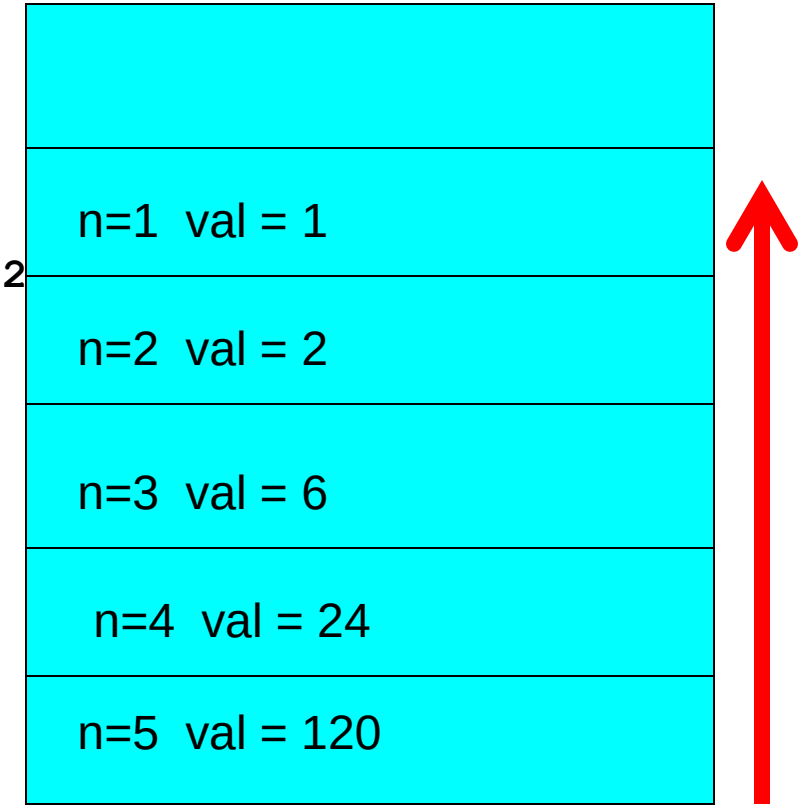
```
#include <stdio.h>
int total; //definiton
extern sum(int a, int b);
main() {
  sum(5,8)
  printf("sum=%d\n", total);
}
```

Local Variables

- Local variables are defined inside functions
- *The Scope* of a Local Variable is from where it is defined to the end of the function.
- *The Lifetime* of a Local Variable is from the time the function starts to the time the function returns.
- Local Variables are stored in the execution stack.
- Local variables are not initialized. The initial variable will be whatever value is in the stack when the function starts.

Local Vars Example

```
#include <stdio.h>

int fact(int n) {
  int val;
  printf("fact(%d)\n", n);
  if (n == 1) {
      mymemdump(stdout, (char*) &n,512
      val = 1;
  }
  else {
        val = n * fact(n-1);
  }
  return val;
}

main()
{
        int v = fact(5);
        printf("fact(5)=%d\n", v);
}
```



Stack Dump of fact(n)

```
0x00007FFFF837243C: 01 00 00 00 28 26 37 F8 FF 7F 00 00 02 07 40 00  ....(&7....@.
0x00007FFFF837244C: 01 00 00 00 80 24 37 F8 FF 7F 00 00 F4 06 40 00  ....$7...@.
0x00007FFFF837245C: 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00  ................
0x00007FFFF837246C: 02 00 00 00 D8 81 37 F8 FF 7F 00 00 68 77 6A CB  ....7..hwj
0x00007FFFF837247C: 95 7F 00 00 B0 24 37 F8 FF 7F 00 00 F4 06 40 00  ..$7...@.
0x00007FFFF837248C: 00 00 00 00 B0 05 40 00 00 00 00 00 F5 13 A6 CB  .....@......
0x00007FFFF837249C: 03 00 00 00 00 00 00 00 00 00 00 00 02 07 40 00  ..............@.
0x00007FFFF83724AC: 00 00 00 00 E0 24 37 F8 FF 7F 00 00 F4 06 40 00  ....$7...@.
0x00007FFFF83724BC: 00 00 00 00 44 09 40 00 00 00 00 00 C0 8C A4 CB  ....D.@.....
0x00007FFFF83724CC: 04 00 00 00 70 BF A5 CB 95 7F 00 00 88 01 C7 CB  ....p...
0x00007FFFF83724DC: 95 7F 00 00 10 25 37 F8 FF 7F 00 00 F4 06 40 00  ...%7...@.
0x00007FFFF83724EC: 00 00 00 00 01 00 00 00 00 00 00 00 FD 08 40 00  ..............@.
0x00007FFFF83724FC: 05 00 00 00 FF B2 F0 00 00 00 00 00 00 00 00 00  ..............
```

The argument n in fact(n) is in red. It shows the different stack frames

Static Local Vars

- If you add the keyword *static* before a local variable it will make the value of the variable be preserved across function invocations.
- The variable will be stored in data/bss instead of the stack.
- The Scope of a static local var is the function it is defined.
- The Lifetime of a static local var is the whole execution of the program.

Example static local var

```
int sum(int a) {
  static int i;
  i += a;
  printf("i=%d\n");
}
main() {
  sum(4);
  sum(5);
  sum(6);
}
Output:
  4
  9
  15
```

String Functions in C

- C Provides string functions such as:
  - strcpy(char *dest, char *src)
    - Copy string src to dest.
  - int strlen(char * s)
    - Return the length of a string
  - char * strcat(char * dest, char *src)
    - Concatenates string src after string dest. Dest should point to a string large enough to have both dest and src and the null terminator.
  - int strcmp(char * a, char * b)
    - Compares to strings a,b. Returns >0 if a is larger alphabetically than b, < 0 if b is larger than a, or 0 if a and ba re equal.
  - See man string

Implementation of strcpy using array operator.

```
char * strcpy(char * dest, char * src) {
    int i = 0;
    while (src[i] != '\0') {
       dest[i] = src[i];
         // Copy character by char from src to dest.
       i++;
    }
   dest[i] ='\0'; // Copy null terminator
   return dest;
}
```

IMPORTANT: dest should point to a block of memory large enough to store the string pointed by dest.

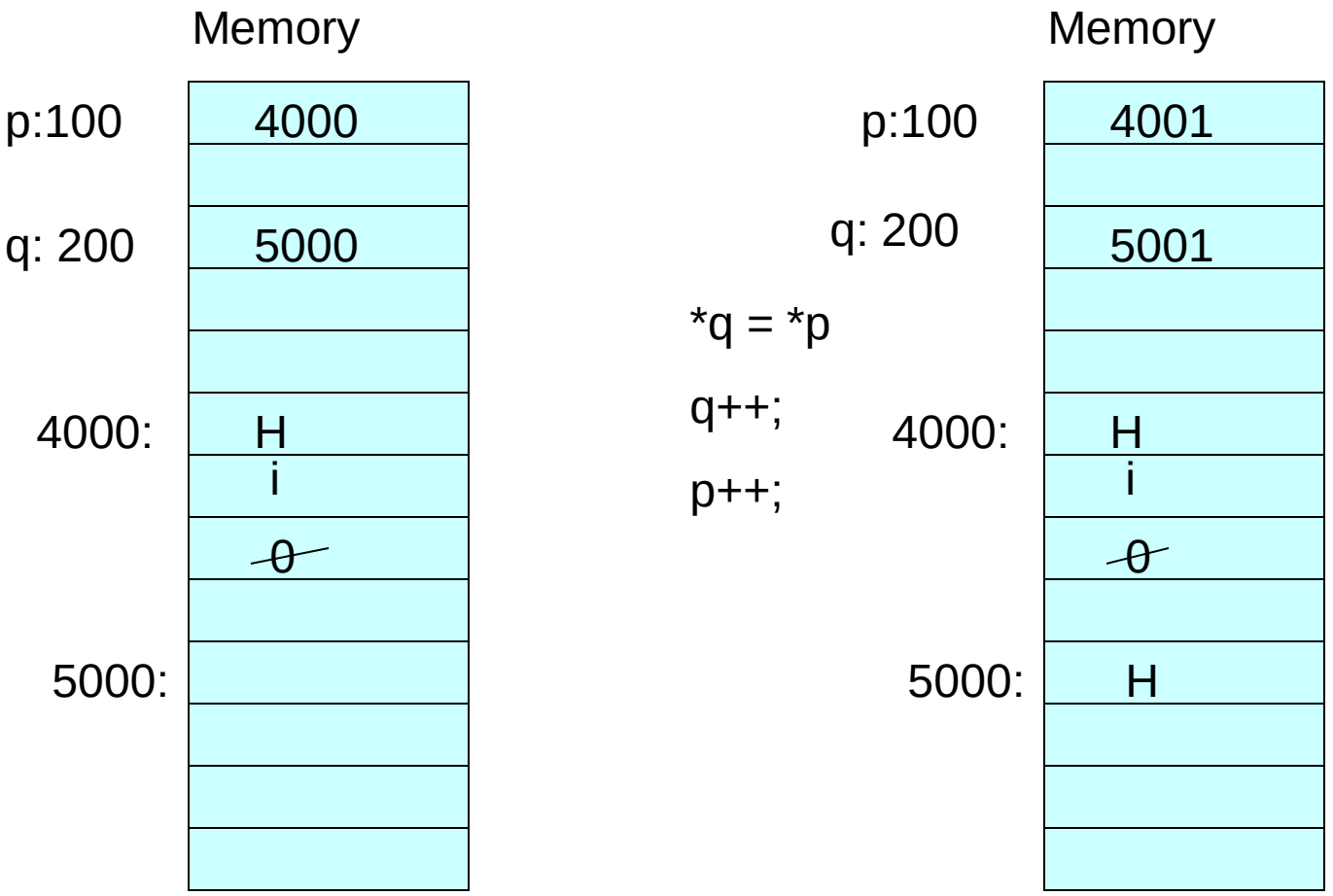
Implementation of strcpy using pointers

Implementation of strcpy using pointer operations.

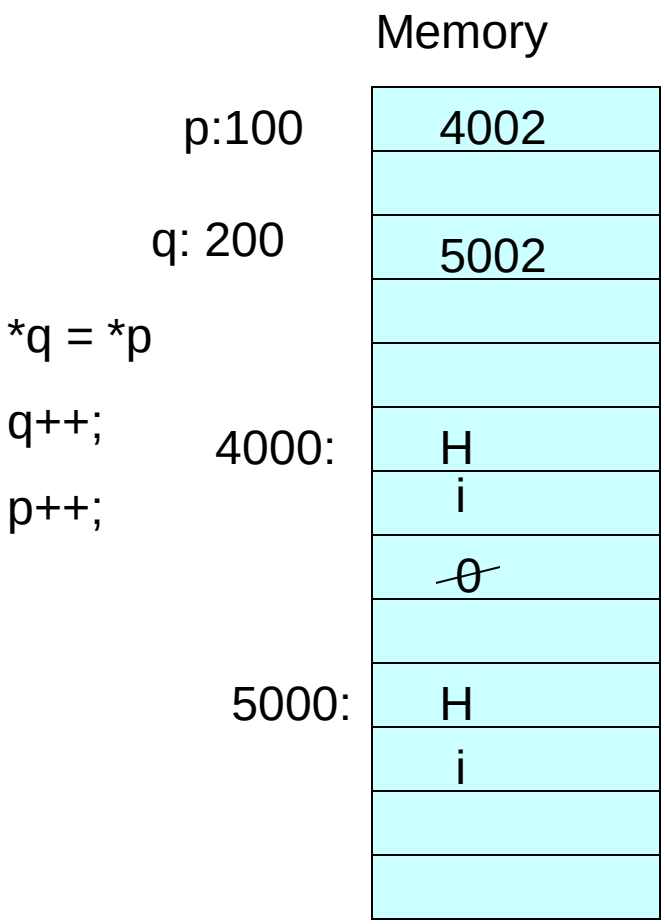
```
char * strcpy(char * dest, char * src) {
    char * p = src;
    char * q = dest;
    while (*p != '\0') {
       *q = *p;
        p++;
       q++;
    }
    *q = '\0';
   return dest;
}
```

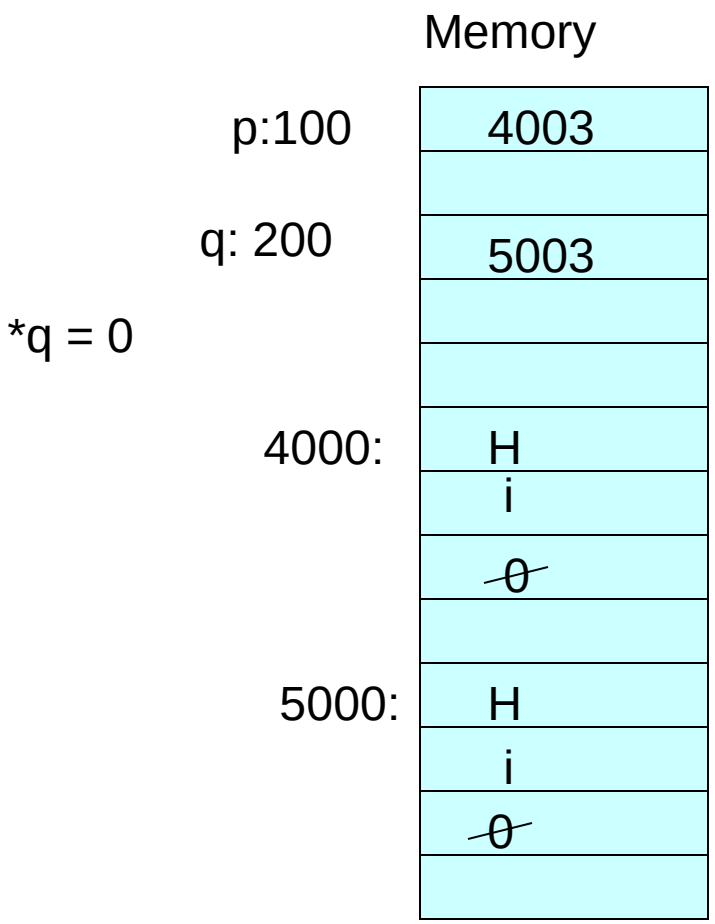
src is a pointer (address) that points to the string to copy.
*src is the character at that location.

Implementation of strcpy using pointers



Implementation of strcpy using pointers





Implementation of strlen

```
char * strlen(char * src) {
    int i = 0;
    while (src[i] != '\0') {
       i++;
    }
    return i;
}
```

Using malloc

- Malloc is used to allocate memory from the system.
- It is similar to “new” in Java but there is no constructor associated to it.
- Example:

```
int *p; // p is a pointer to an int. However, since
        // p is not initialized, it can be
        // pointing anywhere in memory
p = (int*) malloc(sizeof(int)) // Now we can store a value
*p = 5;
```

If we try to store into p without initializing it the program will crash.

Example:

```
int *p;
*p = 5 // ***** CRASH!!!!
```

Allocating an array using malloc

- You may allocate memory for an array in the same way:
- Example:

```
int * array;
int N = 10;
array = (int *) malloc(N*sizeof(int));
// Now array can be used:
array[4] = 10;
```

Allocating an array using malloc

- When memory is no longer in use call free(p);
- If memory is not freed, the memory used by the program will increase and your program will start to slow down.
- Memory that is no longer in use and not freed is called a memory leak.
- Example:

```
int * array = new (N* sizeof(int));
// Use array
free(array);
```

Lexical Structure of C

A program is a sequence of characters in a text file.

```
hello.c
|*......*|
int main()
{
      printf(“Hello World\n”);
}
```

The Compiler groups characters into units called “tokens”(lexical units).

Comments in C

- Comments
- /*.....*/ same as Java
- //..... available in most compilers but it is not in standard.

Identifiers

- Sequence of letters, underscore and digits that do not start with a digit
- Only first 37 chars are significant
- Mix lower and upper case characters to make variable readable

Keywords

- Special keywords.

  while,break, for, case, break,continue, int.....

Types of variables

```
int a;              // Signed Integer 4 bytes
long b;             // Signed integer 8 bytes
short s;            // Signed integer 2 bytes
char c;             // Signed integer 1 byte
unsigned int d;     // Unsigned int 4 bytes
unsigned long e;    // Unsigned int 8 bytes
unsigned short f;   // Unsigned int 2 bytes
unsigned char g;    // Unsigned int 1 byte
```

Types of variables

```
float ff;          // Floating Point number 4 bytes
double dd;         // Floating Point number 8 bytes
long double ddd;   // Floating Point 16 bytes
```

char hello[20]; // A string is an array of chars. Last character is a 0. This string is able to store a string with 19 characters.

Constants

- Same as Java.

```
const double PI = 3.14192654;
```

- constants cannot be changed

- Also it is common in C to use the C pre-processor to define variables

#define PI 3.14192654

Assignment

- The element in the left side has to be an “l-value”. An “l-value” can be interpreted as an address.
- A variable is an l-value but a constant is not an l-value.

  x = 5; ← l-value

  8 = 5; ← not l-value

- In C, an assignment is an expression
- This implies that an assignment may appear anywhere an expression is allowed.

  Example:

```
  j = (i = 5) +3; // this will assign 5 to i and it will assign 8 to j.
```

- Example:

```
  while( (c = getChar() ) != -1){ …
  }
```

The Main Program

- Execution starts in main

  int main(int argc, char**argv){
  }
  or
  int main(){
  }

- argc-store # of args.

- argv-it is an array of the argument entries as strings.

```
int main(int argc, char**argv){
    int i;
    for(i = 0; i <argc; i++){
        printf("argv[%d] = %s\n, i, argv[i]);
    }
}
```

Integer Constants

- Integer Constants

  1 2 3 4 → decimal

  031 → octal constant 3*8+1 = 25 Starts with 0

  0x4A3 → hexadecimal constant 4*16^2+10*16+3=
  or in binary 0100 1010 0011

- The type of integer constant will be:
  - int → if v does not exceed the int range.
  - long → if v exceeds an int but not a long.
  - unsigned long → if v exceeds long

Integer Constants

- You can add suffix to force type
  - 123456789L → long
  - 55u → unsigned int
  - 234Lu → unsigned long

Floating Point Constants

- 3.14 → type is always double
- To force the type float, add “f” suffix.
- 3.14f → float constant

Character constant

- 'q' enclosed with a single quote.
- Also you can use escape sequences with '\octal number' e.g. '\020’

  ascii: 2*8+0=16
- '\hex number with two digits' e.g. '\AE’

  ascii: 10*16+14

Character Constants

- Also there are some common escape sequences '\n' → new line
  - '\r' → return
  - '\t' → tab
  - '\' ' → single quote
  - '\" ' → double quote
  - '\\' → back slash

Character Constants

- Character constants have type int.

```
int i;
i = 'A'; //assign ascii 65 to I
Or
i = 65;
printf("A = %d\n", 'A');
```

output
A=65

- Example:

```
//check if a letter is a lowercase
   if(c >= 'a' && c <= 'z'){
        printf("%c char %d is lower case \n", c, c);
   }
```

String Constants

- “My String” is a string constant of type (char *)
- There are no operations with string in Java like “Hello”+”world”.
- However, two consecutive string constants can be put together by the complier

  `“Hello. ” “world”` is equivalent to

  `“Hello. world”`
- So you can have multi-line strings like,

  char * class =

  “CS240 \n”

  “Programming in C \n”;
- The compiler will put both constants in a single string.

Short-Circuit && (and) and || or expression

- ***e1 && e2*** is the short circuit "and" expression
  - If e1 is false, e2 is not evaluated.

```
if(x && (i = y))
//  If x is false
// then i=y is never evaluated.
```

- ***e1 || e2*** is the short circuit "or" expression
  - If e1 is true, then e2 is never evaluated

```
if (x || (i=y))
  // If x is true, then i=y
  // is never evaluated.
```

Boolean and Int

- There is no Boolean type
- A 0 is False and anything different than 0 is True.
- if(5){
  //always executed
  }
  if(0){
  //never executed
  }

Conditional Expressions

- e = (e1 ? e2 : e3) *conditional expression

Equivalent to

```
if(e1){
    e = e2;
}
else{
    e = e3;
}
```

- Example:

```
x = 3 + ( ( b == 4 ) ? 7 : 8 )
```

Equivalent to:

```
if(b==4){
    x = 3+7;
}
else{
    x=3+8;
}
```

Comma Expressions

- i = (e1, e2) *comma expressions
  - the value of i is the last expression e2.
- i=(e1, e2, e3)
  - the value of i is the last expression e3
- Useful for the “for” statement to execute multiple increments or assignments.
- Example:

  for(i=0, j=3; i<7; i++, j—) {…}

Arithmetic Conversion

- If operands in an expression have different types the operands will have their types changed from the lower precision type to the higher precision type
- → 5 / 2 = 2 types is an int

  (int) (int) division of 2 ints
- → 5 / 2.0 = the compiler evaluates 5 and convert it to 5.0(double) = 2.5 type is double

  (int) (double) division between int and double

- (1 / 2) * (3.0 + 2) = 0

use (1.0 / 2) * (3.0 + 2) instead so the result is 2.5 double

- C does not perform arithmetic at precision smaller than int

Arithmetic Conversion

- *int i;*
  i = 'A' + 'B'; → 131 (int)
- Even though 'A' and 'B' is char it is converted to int (65 + 66)
- - A lower precision type converted to a higher precision type
- int → unsigned → long → unsigned → float → double → long double
- *int i = 10;*
  unsigned u = 20;
- *i + u* -> i is converted to unsigned since unsigned has more precision to int

Assigment conversion

- They happen when an expression in the right hand side has to be converted to the type in the left hand side in an assignment.

```
int i = 3;
double d;
d = i;  // i is converted to double
```

- You have to be careful if you are assigning to a variable with a smaller precision

```
d = 2.5;
i = d;  // 2.5 converted from double to int 2
```

- You will get a warning and some cases an error. Use a cast instead

```
i = (int)d;
```

Cast Conversion

- Done by the programmer

  (type) expression

  (int) 2.5 results into 2

typedef

- typedef provides a synonym of an existing type

```
        typedef int Boolean;
        Boolean b;
#define FALSE 0
#define TRUE  1
        b = 1;
        b = FALSE;
        b = TRUE;
```

Common Errors

- “a” and ‘a’ are different
  - “a” is a string constant type (char *)
  - ‘a’ is a char constant type (int)

- `if (c = getchar() != EOF)`

  is not the same as

  `if ((c = getchar()) != EOF)`

if Statement

- if(..expression..){
  .....
  .....
  }
- Expression in if (exp) statement is converted to int.
- If (expression) != 0 then expression is true and the body of the statement is executed.
- if(expression) == 0 then it is false and the body is not executed. Then continues to check against else if or else statements are used to then be executed.
- Example:

```
int i;
if(i != 0){
      // do something
}
```

- equivalent to:

```
if(i){
      // do something
}
```

while statement

- while(..expression..){
  .....
  .....
  }
- Expression in while statement is converted to an int.

- while(expression) != 0 then the program continuous to loop until expression does equal 0.
- Example:

```
int i, j;
while(i > j){
    ....          // code body that will continue to executed
                  // until j >= i
}
```

for statement

- "for" statement is typically used in situation where you know in advance the number of iterations.
- Syntax:

```
for(expression1; expression2; expression3){
     ....
}
```

- Example:

```
//Assuming the variable i has been declared above.
for(i = 0; i < 10; i++){
  ....... // code to be executed goes here
          // this specific for statement will
          // loop until !(i < 10) for a total
          // of 10 iterations
}
```

for statement

- However you could use the for statement where a while statement is also used.

  Example:

```
expr1;
while(expr2){
    // body of statement to be executed
    expr3;
}
```

  expr1 and expr3 are usually 'coma expressions'

```
        expr1        expr2     expr3
for(  i = 0, j = 0;   i < 10;     i++    ){
   // body of statement to be executed
}
```

switch statement

- switch(expr) {
  case const1: .... break;
  case const2: .... break;
  default: .... break;
  }
- expr is evaluated and converted to an int.
- If ....expr.... is equal to any of const values that block of code is executed.
- Default is evaluated if expr does not match any of the case consts.

Forever Loops (Infinite Loops)

- ```
  while(1){
      // body runs forever
  }
  ```

- ```
  for(;;){
      // body runs forever
  }
  ```

Example: Count the number of lines, tabs and lower case characters in the input

```
#include <stdio.h>

int main(){
  int countBlanks = 0;
  int countTabs = 0;
  int countNewline = 0;
  int countLower = 0;
  int c;
```

Example: Count the number of lines, tabs and lower case characters in the input

```
while((c = getchar()) != EOF){
     switch(c){
             case ' ':
                     countBlank++;
                     break;
             case '\t':
                     countTabs++;
                     break;
             case '\n':
                     countNewline++;
                     break;
             default:
                     if(c >= 'a' && c <= 'z'){
                             countLower++;
                     }
                     break;
     }       //End of switch statement
} //End of while loop
```

Example: Count the number of lines, tabs and lower case characters in the input

```
    //Within a switch statement when a "break"
    //is hit it exits the switch statement.
   //If there is no break statement within a
  // case the program would then continue to
  // execute each case until a break statement is hit.

  printf("blanks=%d tabs=%d newlines=%d lower= %d\n",
     countBlanks, countTabs, countNewLines, countLower);
}  //End of main
```

Text Files

- Declared in the header
  ```
  #include <stdio.h>
  ```
- You need a file handle to use files
  ```
  FILE *fileHandle;
  ```
- To open a file you would use:
  ```
  fileHandle = fopen(fileName, fileMode);
  ```
- fileMode

  "r" -> open file for reading

  "w" -> open file for writing

  “a" -> open file for appending. Created if it does not exist.

  "r+"-> open file for both read and write. But the file must exist.

  "w+"-> open file for both read and write. If file exists it is overwritten, if !exists then the file is created

  “a+"-> open file for reading or appending, if does not exist file is created.
- fopen will return NULL if it fails.

Example of fopen

```
FILE *f;
f = fopen("hello.txt", "r");
if(f == NULL){
   printf("Error cannot open hello.txt\n");
   perror("fopen");  // print why last called failed
   exit(1);          // passing the value of 1 to exit
                     // means that the reason for the
                     // sudden exit was because of
                     // an error.
}

// Read file using fscanf, fgetc or fread.

// close the file
fclose(fileHandle);
```

Standard Files

- There are three FILE's that are already opened when a program starts running:
- stdin - standard input, It is usually the terminal unless input is redirected:
  - bash$: hello (stdin is terminal)
  - bash$: hello < in1.txt (stdin is file in1.txt)
- stdout: standard output. It is usually the terminal unless redirected to output to a file:
  - bash$: hello (stdout is terminal)
  - bash$: hello >> out.txt (stdout is out.txt. Output is appended)
- stderr: Errors are redirected to stderr. It is usually the terminal unless redirected to a file.
  - bash$: hello > out 2>&1 (Redirect both stdout and stderr to out)

Basic Operations for stdin/stdout

- For stdout:
  - int printf(....) prints formatted output to stdout
  - int putchar(int c) writes c to stdout
- For stdin
  - int getchar() reads one character at a time
  - int scanf(.....) reads formatted input from stdin -

Basic Operations for generic FILE's

- int fgetc(fileHandle)
  - reads one char from fileHandle
  - c = getchar is equivalent to c = fgetc(stdin)
- int fputc(fileHandle, c)
  - puts/output char to stdout
  - putchar(c) is equivalent to fputc(stdout,c)
- int fscanf(fileHandle, format, &var….)
  - read formatted input from file handle,
  - scanf("%d", &j); is equivalent to fscanf(stdin, "%d", &i);

Basic Operations for generic FILE's

- int fprintf(fileHandle, format, ...);
  - prints formatted output to fileHandle
  - printf("Hello world %d\n, i); is equivalent to fprintf(stdout, "Hello world%d\n", i);
- sprintf(char *str, format, ...);
  - equivalent to printf/fprintf but the output is a string
  - "str" is passed as argument. str should have enough space to store the output.

Example: Read file with student grades and compute average.

students.txt:

```
Mickey 91
Donald 90
Daisy 92
```

students.c

```
#include<stdio.h>
#include<stdlib.h>   // perror, exit and others

#define MAX_STUDENTS 100
#define MAX_NAME     50
#define STUDENTS_FILE        "students.txt"

char names[MAX_STUDENTS][MAX_NAME + 1];
int grades[MAX_STUDENTS];
```

Example: Read file with student grades and compute average.

```
int main(int argc, char **argv){
      FILE *fd;
      char name[MAX_NAME + 1];
      int grade;
      int n;
      int i;
      double sum;

      fd = fopen(STUDENTS_FILE);

      if(fd == NULL){
            printf("cannot read %s\n", STUDENT_FILE);
            perror("fopen");
            exit(1);
      }
```

Example: Read file with student grades and compute average.

```
    n = 0;
    while(fscanf(fd, "%s %d",
          names[n], &grades[n]){
          n++;
    }

    //computer average sum
    for(i = 0; i < n; i++){
          sum += grades[i];
    }
    printf("Average: %lf\n", sum / n);
    fclose(fd);
}
```

Lab3: Implementing resizable table

resizable_table.h

```
#if !defined RESIZABLE_ARRAY_H
#define RESIZABLE_ARRAY_H

#define INITIAL_SIZE_RESIZABLE_TABLE 10

typedef struct RESIZABLE_TABLE_ENTRY {
        char * name;
        void * value;
} RESIZABLE_TABLE_ENTRY;

typedef struct RESIZABLE_TABLE {
        int maxElements;
        int currentElements;
        struct RESIZABLE_TABLE_ENTRY * array;
} RESIZABLE_TABLE;

RESIZABLE_TABLE * rtable_create();
int rtable_add(RESIZABLE_TABLE * table, char * name, void * value);
…
#endif
```

Lab3: Implementing a resizable table

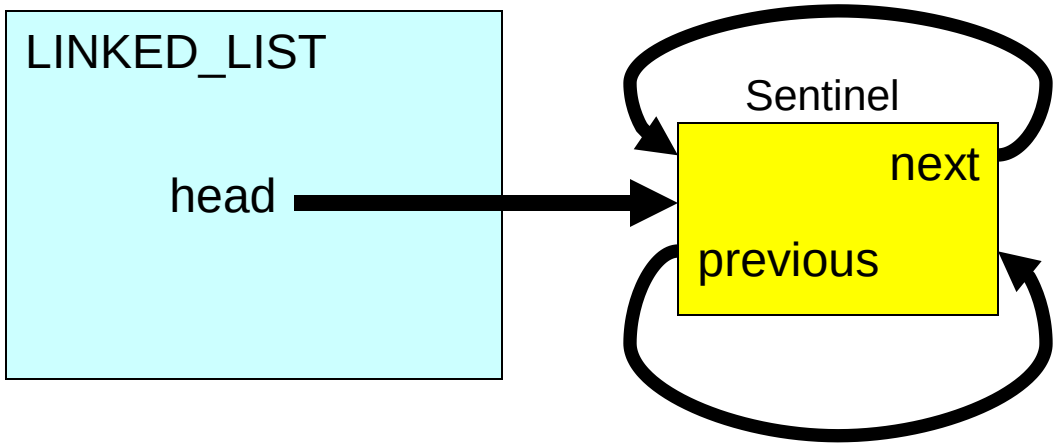
resizable_table.cpp:

```
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include "resizable_table.h"

//
// It returns a new RESIZABLE_TABLE. It allocates it dynamically,
// and initializaes the values. The initial maximum size of the array is 10.
//
RESIZABLE_TABLE * rtable_create() {

        // Allocate a RESIZABLE_TABLE
        RESIZABLE_TABLE * table = (RESIZABLE_TABLE *) malloc(sizeof(RESIZABLE_TABLE));
        if (table == NULL) {
                return NULL;
        }
        // Initialze the members of RESIZABLE_TABLE
        table->maxElements = INITIAL_SIZE_RESIZABLE_TABLE;
        table->currentElements = 0;
        table->array = (struct RESIZABLE_TABLE_ENTRY *)
                malloc(table->maxElements*sizeof(RESIZABLE_TABLE_ENTRY));
        if (table->array==NULL) {
                return NULL;
        }

        return table;
}
```

Lab3: Implementing a resizable table

```
//
// Adds one pair name/value to the table. If the name already exists it will
// Substitute its value. Otherwise, it will store name/value in a new entry.
// If the new entry does not fit, it will double the size of the array.
// The name string is duplicated with strdup() before assigning it to the
// table.
//
int rtable_add(RESIZABLE_TABLE * table, char * name, void * value) {

        // Find if it is already there and substitute value

        // If we are here is because the entry was not found.

        // Make sure that there is enough space

        //
        // Add name and value to a new entry.
        // We need to use strdup to create a copy of the name but not value.
        //

        return 0;
}
```

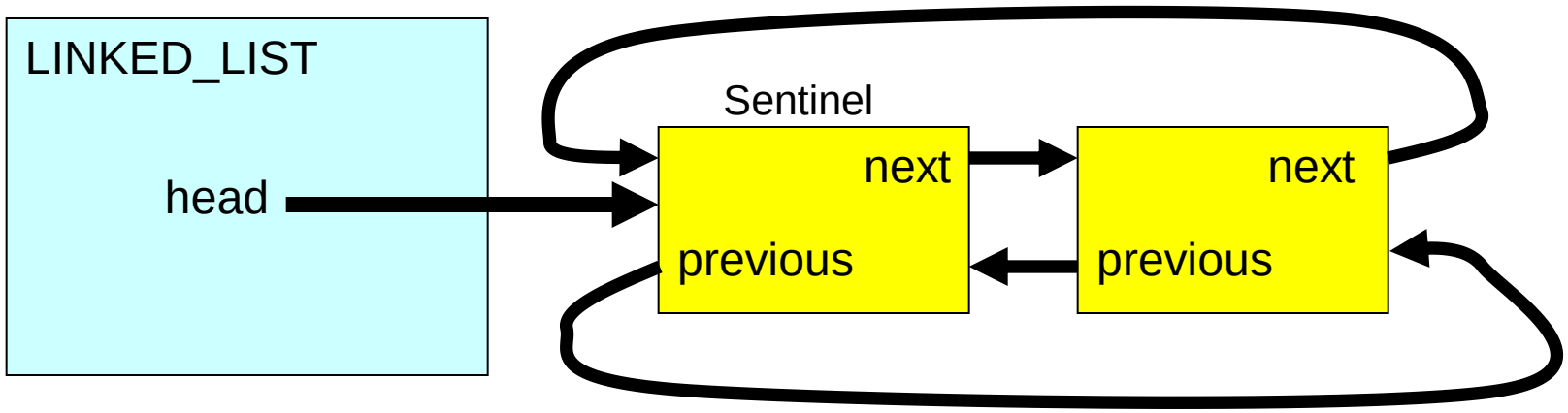
Implementing a Double Linked List



Sentinel – It does not store any data but simplifies implementation.

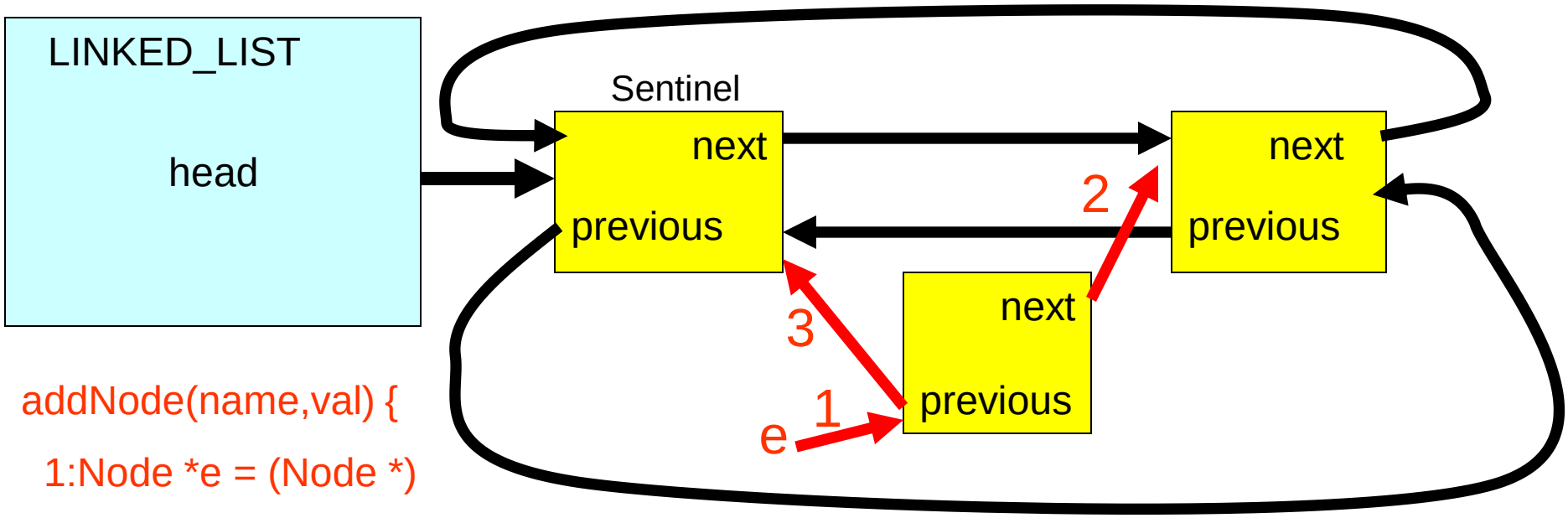
Empty List

Implementing a Double Linked List



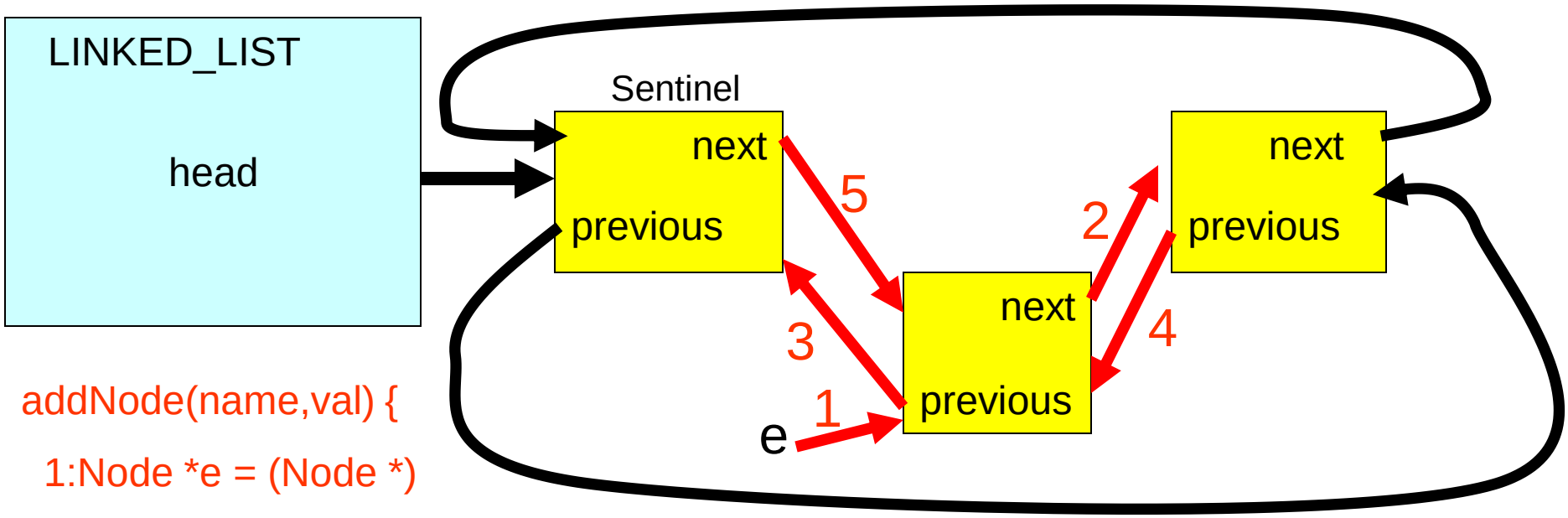
List with one element

Adding a Node to a Double Linked List




```
addNode(name,val) {
  1:Node *e = (Node *)
       malloc(sizeof(Node));
  e->name = strdup(name)
  2: e->next = head->next;
  3: e->previous = head;
```

Adding a Node to a Double Linked List




```
addNode(name,val) {
  1:Node *e = (Node *)
       malloc(sizeof(Node));
  e->name = strdup(name)
  e->val = val
  2: e->next = head->next;
  3: e->previous = head;
  4:e->next->previous = e;
  5: head->next = e;
}
```

Lab3: Implementing a double-linked list

linked_list.h:

```
#if !defined LINKED_LIST_H
#define LINKED_LIST_H

typedef struct LINKED_LIST_ENTRY {
        char * name;
        char * value;
        struct LINKED_LIST_ENTRY * next;
        struct LINKED_LIST_ENTRY * previous;
} LINKED_LIST_ENTRY;

typedef struct LINKED_LIST {
        int nElements;
        LINKED_LIST_ENTRY * head;
} LINKED_LIST;

LINKED_LIST * llist_create();
void llist_print(LINKED_LIST * list);
int llist_add(LINKED_LIST * list, char * name, char * value);
char * llist_lookup(LINKED_LIST * list, char * name);
int llist_remove(LINKED_LIST * list, char * name);
int llist_get_ith(LINKED_LIST * list, int ith, char ** name, char ** value);
…
#endif
```

Lab3: Implementing a double-linked list

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "linked_list.h"
//
// It returns a new LINKED_LIST. It allocates it dynamically,
// and initializaes the values. The initial list is empty.
//
LINKED_LIST * llist_create() {

        LINKED_LIST * list = (LINKED_LIST *) malloc(sizeof(LINKED_LIST));
        if (list == NULL) {
                return NULL;
        }

        // Create Sentinel node. This node does not store any data
        // but simplifies the list implementation.
        list->head = (LINKED_LIST_ENTRY *) malloc(sizeof(LINKED_LIST_ENTRY));
        if (list->head == NULL) {
                return NULL;
        }


        list->nElements = 0;
        list->head->next = list->head;
        list->head->previous = list->head;

        return list;
}
```

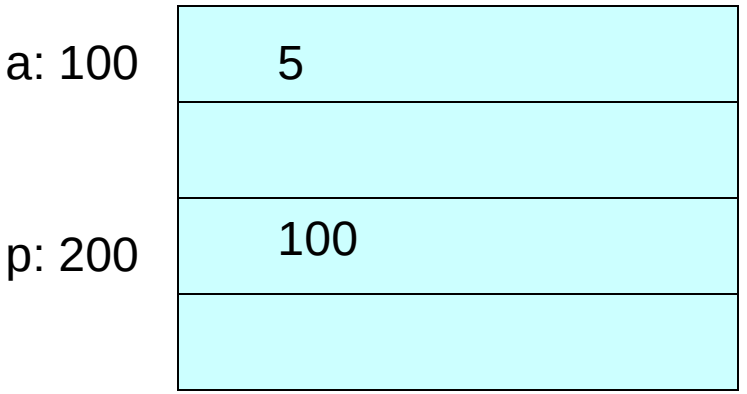
Lab3: Implementing a double-linked list

```
//
// Adds one pair name/value to the list. If the name already exists it will
// Substitute its value. Otherwise, it will store name/value in a new entry.
// The name/vale strings are duplicated with strdup() before adding them to the
// list.
//
int llist_add(LINKED_LIST * list, char * name, char * value) {
        // TODO: Search if entry is not in the list. If it is then substitute value and return.

        // Entry not found. Add new entry at the end of the list
        e = (LINKED_LIST_ENTRY *)malloc(sizeof(LINKED_LIST_ENTRY));
        e->name = strdup(name);
        e->value = strdup(value); // We do this only in the linked-list. The value is always char *.

        //Add at the beginning
        e->next = list->head;
        e->previous = list->head->previous;
        e->next->previous = e;
        e->previous->next = e;

        list->nElements++;

        return 0;

}

//
// Returns the value that correspond to the name or NULL if the
// name does not exist in the list.
//
char * llist_lookup(LINKED_LIST * list, char * name) {
        return NULL;
}

//
// It removes the entry with that name from the list.
// Also the name and value strings will be freed.
//
int llist_remove(LINKED_LIST * list, char * name) {
        return 0;
}
```

Pointers

A pointer is an address in memory.

```
int a;
int *p;
a=5;
p=&a;
```




```
printf("a= %d",a);   // prints 5
printf("Address of a = %ld",&a); // prints 100
printf("p=%ld",p);  //prints 100
printf("p=%ld",&p);  //prints 200
printf("Value *p = %d",*p);   //prints 5
```

Pointers

A pointer is an address in memory.

```
int a;
int *p;
p=&a;
*p =8
```

Printing Pointers

- * is “value at” operator
- To print pointers we often use the “0x%lx” format since %ld may print pointers as negative numbers
- If you want to print pointers as unsigned decimals, use “%lud” or just “%lu” (will print positive number)

Pointer Types

- Pointers have types :

```
int i;
int * p;   // p is an integer pointer
unsigned char  *q; // q is a pointer to unsigned char
double * pd;  // pd is a pointer to a double
```

- Sometimes it is necessary to convert one pointer type to another:

```
q = (unsigned char *) p;
```

- This allows to store in or read from the same memory but as a different type.

Little Endian /Big Endian

- Assume the following statements:

```
int i = 5;
unsigned char * p;
p = (unsigned char*) &i;
```

- 5 may be stored in two ways:

Little Endian:

Least Significant Digit in Smallest Memory Location

i: 100	5
101:	0
102:	0
103:	0

Big Endian:

Least Significant Digit in Smallest Memory Location

i: 100	0
101:	0
102:	0
103:	5

Code to Determine if machine is Little Endian

```
Int isLittleEndian()
{
  int i = 5;
  unsigned char * p = (unsigned char *) &i;

  if (p[0] == 5) {
    return 1;
  }
  else {
    return 0;
  }
}
```

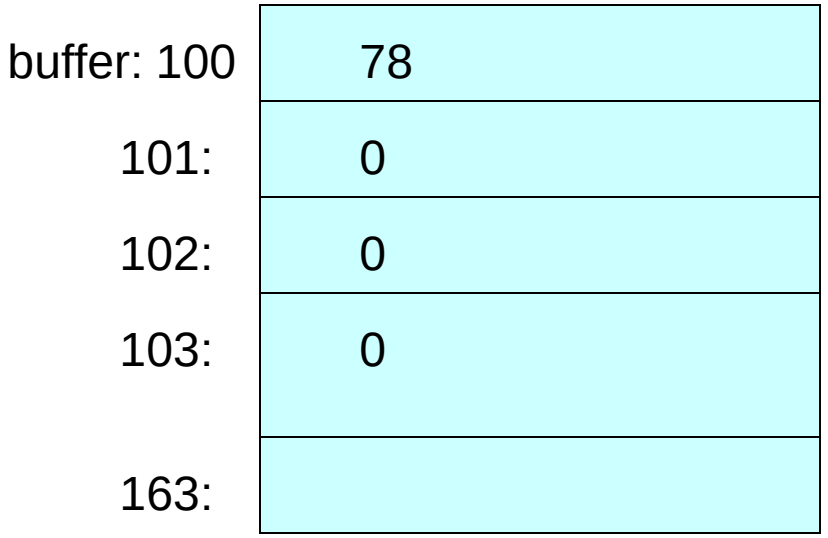
Little Endian /Big Endian

- **Little Endian** : Puts 5 at 100 (Intel, ARM, VAX)
  - Lowest significant byte is placed in the lowest address
- **Big Endian** : Puts 5 at 103 (Sparc, Motorola)
  - Lowest significant byte in the highest address

Pointer Conversion

- Pointer conversion is very powerful because you can read or write values of any type anywhere in memory.

```
char buffer[64];
int *p;
p=(int *)&buffer[20];
*p = 78;
```



Malloc and Dynamic Memory

- malloc() allows us to request memory from the OS while the program is running.
- Instead of pre-allocating memory (eg. Maximum matrix) when the program is compiled, we can allocate it at runtime using malloc()

  p=(T*) malloc(sizeof(T));

- This allocates memory of sizeof(T) bytes and assigns it to ‘p’ (after casting)
- p=calloc(1,sizeof(T)) allocates memory for an object of type T like in malloc
- The only difference is that calloc() initializes memory to 0.
- calloc() calls malloc() internally and then initializes memory to 0

Example Using malloc

```
int *p;
p=(int *) malloc(sizeof(int));
if(p==NULL) {
  //Not enough memory
  perror("malloc");
  exit(0);
}
*p = 5;

// malloc may return NULL if there is not
// enough memory in the process.
```

Memory Deallocation

- When the program does not need the memory anymore, you can free it to return it to the malloc free list.

```
free(p);
```

- The free list contains the list of objects that are no longer in use and that can be allocated in future malloc() calls.
- There is no garbage collection in C.

Memory Allocation Errors

- Explicit Memory Allocation (calling free) uses less memory and is faster than Implicit Memory Allocation (GC)
- However, Explicit Memory Allocation is Error Prone
  1. Memory Leaks
  2. Premature Free
  3. Double Free
  4. Wild Frees
  5. Memory Smashing

Memory Leaks

- Memory leaks are objects in memory that are no longer in use by the program but that ***are not freed***.
- This causes the application to use excessive amount of heap until it runs out of physical memory and the application starts to swap slowing down the system.
- If the problem continues, the system may run out of swap space.
- Often server programs (24/7) need to be “rebounced” (shutdown and restarted) because they become so slow due to memory leaks.

Memory Leaks

- Memory leaks is a problem for long lived applications (24/7).
- Short lived applications may suffer memory leaks but that is not a problem since memory is freed when the program goes away.
- Memory leaks is a “slow but persistent disease”. There are other more serious problems in memory allocation like premature frees.

Memory Leaks

Example:

```
int * ptr;
ptr = (int *) malloc(sizeof(int));
*ptr = 8;
… Use ptr …
ptr = (int *) malloc(sizeof(int));
// Old block pointed by ptr
// was not deallocated.
```

Premature Frees

- A premature free is caused when an object that is still in use by the program is freed.
- The freed object is added to the free list modifying the next/previous pointer.
- If the object is modified, the next and previous pointers may be overwritten, causing further calls to malloc/free to crash.
- Premature frees are difficult to debug because the crash may happen far away from the source of the error.

Premature Frees

Example:

```
int * p = (int *) malloc(sizeof(int));
* p = 8;
free(p); // delete adds object to free list
         // updating header info
…
*p = 9;  // next ptr will be modified.
… Do something else ...
int *q = (int *) malloc(sizeof(int));
 // this call or other future malloc/free
 // calls will crash because the free
 // list is corrupted.
 // It is a good practice to set p = NULL
// after delete so you get a SEGV if
// the object is used after delete.
```

Premature Frees. Setting p to NULL after free.

- One way to mitigate this problem is to set to NULL the ptr after you call free(p)

```
int * p = (int *) malloc(sizeof(int));
* p = 8;
free(p); // delete adds object to free list
P = NULL; // Set p to NULL so it cannot be used
*p = 9;  // You will get a SEGV. Your program will
  crash ad you will know that you have already
  freed p.
```

Double Free

- Double free is caused by freeing an object that is already free.
- This can cause the object to be added to the free list twice corrupting the free list.
- After a double free, future calls to malloc/free may crash.

Double Free

Example:

```
int * p = (int *) malloc(sizeof(int));
free(p); // delete adds object to free list
.. Do something else ….
free(p); // deleting the object again
         // overwrites the next/prev ptr
         // corrupting the free list
         // future calls to free/malloc
         // will crash

// Also to prevent this problem you may set p
   to NULL after free.
```

Double Free. Setting p to NULL after free.

```
int * p = (int *) malloc(sizeof(int));
free(p); // delete adds object to free
  list
P = NULL;
.. Do something else ….
free(p); // deleting the object again
         // SEGV. And the you can
         // see the stack trace
```

Wild Frees

- Wild frees happen when a program attempts to free a pointer in memory that was not returned by malloc.
- Since the memory was not returned by malloc, it does not have a header.
- When attempting to free this non-heap object, the free may crash.
- Also if it succeeds, the free list will be corrupted so future malloc/free calls may crash.

Wild Frees

- Also memory allocated with *malloc()* should only be deallocated with *free()* and memory allocated with new should only be deallocated with delete.
- Wild frees are also called “free of non-heap objects”.

Wild Frees

Example:

```
int q;
int * p = &q;

free(p);
 // p points to an object without
 // header. Free will crash or
 // it will corrupt the free list.
```

Wild Frees

Example:

```
char * p = (char*)malloc(100);
p=p+10;

free(p);
// p points to an object without
// header. Free will crash or
// it will corrupt the free list.
```

Memory Smashing

- Memory Smashing happens when less memory is allocated than the memory that will be used.
- This causes overwriting the header of the object that immediately follows, corrupting the free list.
- Subsequent calls to malloc/free may crash
- Sometimes the smashing happens in the unused portion of the object causing no damage.

Memory Smashing

Example:

```
char * s = (char*)malloc(8);
strcpy(s, "hello world");

// We are allocating less memory for
// the string than the memory being
// used. Strcpy will overwrite the
// header and maybe next/prev of the
// object that comes after s causing
// future calls to malloc/free to crash.
// Special care should be taken to also
// allocate space for the null character
// at the end of strings.
```

Debugging Memory Allocation Errors

- Memory allocation errors are difficult to debug since the effect may happen farther away from the cause.
- Memory leaks is the least important of the problems since the effect take longer to show up.
- As a first step to debug premature free, double frees, wild frees, you may comment out free calls and see if the problem goes away.
- If the problem goes away, you may uncomment the free calls one by one until the bug shows up again and you find the offending free.

Debugging Memory Allocation Errors

- There are tools that help you detect memory allocation errors.
  - IBM Rational Purify
  - Bounds Checker
  - Insure++
  - Valgrind
  - Dr. Memory

Common Errors with Pointers

- Use a pointer without initializing it (crashes the program SEGV)
- Not allocation enough memory

```
int *array; int n=20;
array=(int *)malloc(sizeof(n * sizeof(int) );
                // 20 * 4 bytes = 80 bytes allocated
array[5]=20;    //OK
array[25]=7;    //C blindly tries to assign 7 to the
                // 25th position, which does not
                // exist as valid memory
```

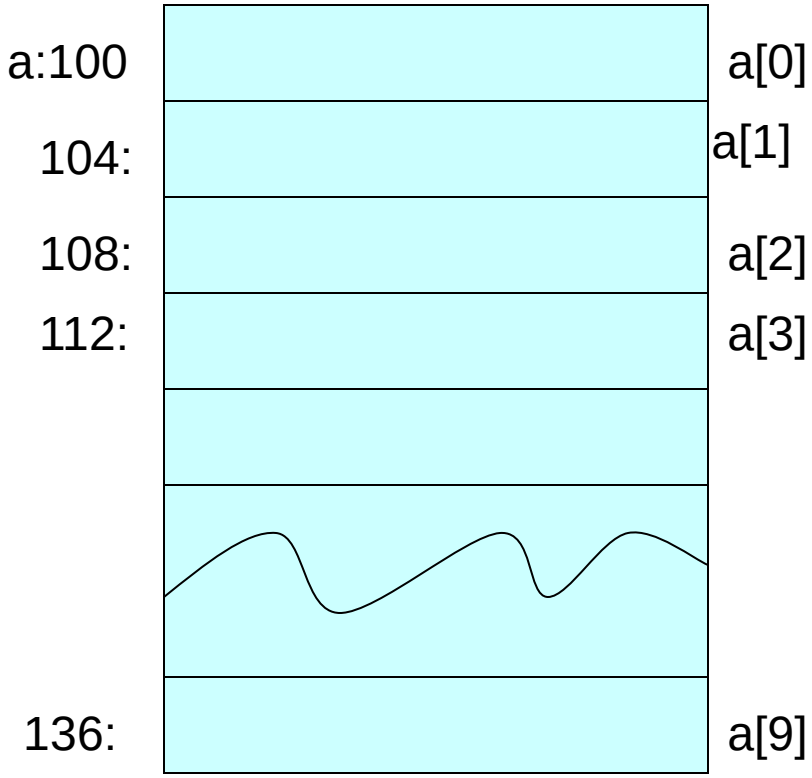
Sum of a pointer and an int

- A pointer is an address when you add an integer i to a pointer of type T, the integer is multiplied by the size of the type T and added to the pointer.

```
int a[10];
int *p;
p = &a[0];  // Assume p is 100
p = p + 1; // Since p is of type int the
           // constant 1 will be multiplied
           // by sizeof(int) before adding.
           // p now is 104 and points to a[1].

p = p + 2; // p is now 104 + 2 * sizeof(int)) =
       // 104 + 2 * 4 = 112 which points to a[3]
```

Pointers and Arrays

int a[10];




```
int a[10];
int *p;
p = &a[0];  // p == 100
p = p + 1;  // Since p is of type int the
            // constant 1 will be multiplied
            // by sizeof(int) before adding.
            // p is now  104 and points to a[1].

p = p + 2;
    // p is now 104 + 2 * sizeof(int)) =
    // 104 + 2 * 4 = 112 which points to a[3]
```

Pointers And Arrays

- In C, pointers are arrays and arrays are pointers.

  For example,

  int a[10];

- ***a*** is an array of 10 elements of type int.
- also, ***a*** is a pointer to the first element of the array

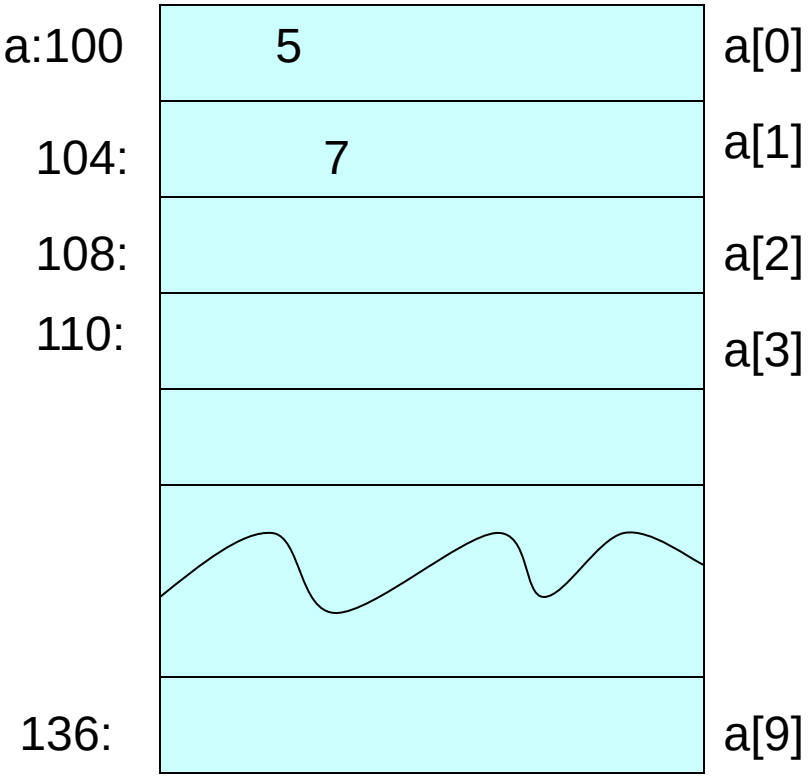
  a is the same as &a[0]

  a[0] is the same as *a

  a[1] is the same as *(a+1)

Pointers and Arrays

int a[10];



a[1]=7
a + 1 == 104
*(a+1) == 7

Pointer Equivalence

- Assume that

  int a[10];

  int i;

- We have that

  a[i] is the same as

  *(a+i) is the same as

  *(&a[0]+i) is the same as

  (int)((char*)&a[0] + i*sizeof(int))

Pointer Comparison

(< >= >= == !=)

- You can compare two pointers.
- A pointer is an unsigned long that is the address of the value it is pointing to
- for example add the elements of an array of integers

```
int sum (int *a, int n) {
    int s = 0;
    for (int  i = 0 ; i < n ; i++) {
         s += a[i];
         }
         return s;
   }
```

Add Elements in an Array Using Pointers

```
int sum (int *a, int n) {
        nt *p;
        int *pend;
        p = &a[0];
        pend = p + n;
        int s = 0;
        while (p < pend){
                        sum += *p;
                        p++; // point p to the next element
        }
        return s;
}
```

- The second part using pointers is faster than the one using arrays because array indexing a[i] requires multiplication.
- a[i] is computed as *(int*)((char*)a+ i * sizeof(int))

Pointers as arrays

By the same token, pointers can be treated as arrays

```
int *p;
p = a;
a[0] = 5;
printf("p[0] = %d\n", p[0]);

Output:
p[0] = 5;

a[7] = 19;
printf("p[7] = %d\n", p[7]);

Output:
p[7] = 19;
```

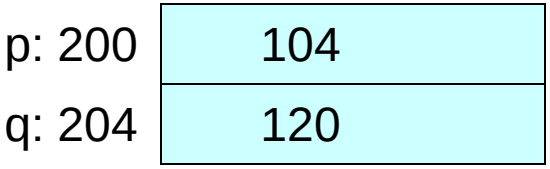
Pointer subtraction

- Assuming that p and q are pointers of the same type, (q – p) will give the number of objects between q and p.

```
int *p, *q;
int buffer[20];
p = &buffer[1];
q = &buffer[5];
printf("q - p =%d\n",q-p);
// it is going to print 16 / 4 = 4
```

Pointer Subtraction

p = &buffer[1];
q = &buffer[5];



q-p == (120-104)/4 ==4

printf("q - p =%d\n",q-p); //Prints 4

Passing Arguments by Value

- All arguments in a function in C are passed by a “value”, that is the value of the variable or constant is passed to the function

```
void a (int x)
        printf(“x=%d\n”,x);
        x = x+1;
}
...
int y = 5;
a(y); // output: x = 5
printf(“y = %d\n”,y); // output: y = 5
```

- The reason that y gets the value of 5 even when x = x+1 in a, is because in a, only the value of y that is 5 is passed.
- Internally, x and y are 2 different variables.

Passing an Argument by Reference

- Passing by reference means that the variable passed in the argument can be modified inside the function.
- This is emulated by passing a pointer as a variable and treating the variable as a pointer.

Example: we want to print the value of the variable and increase it.

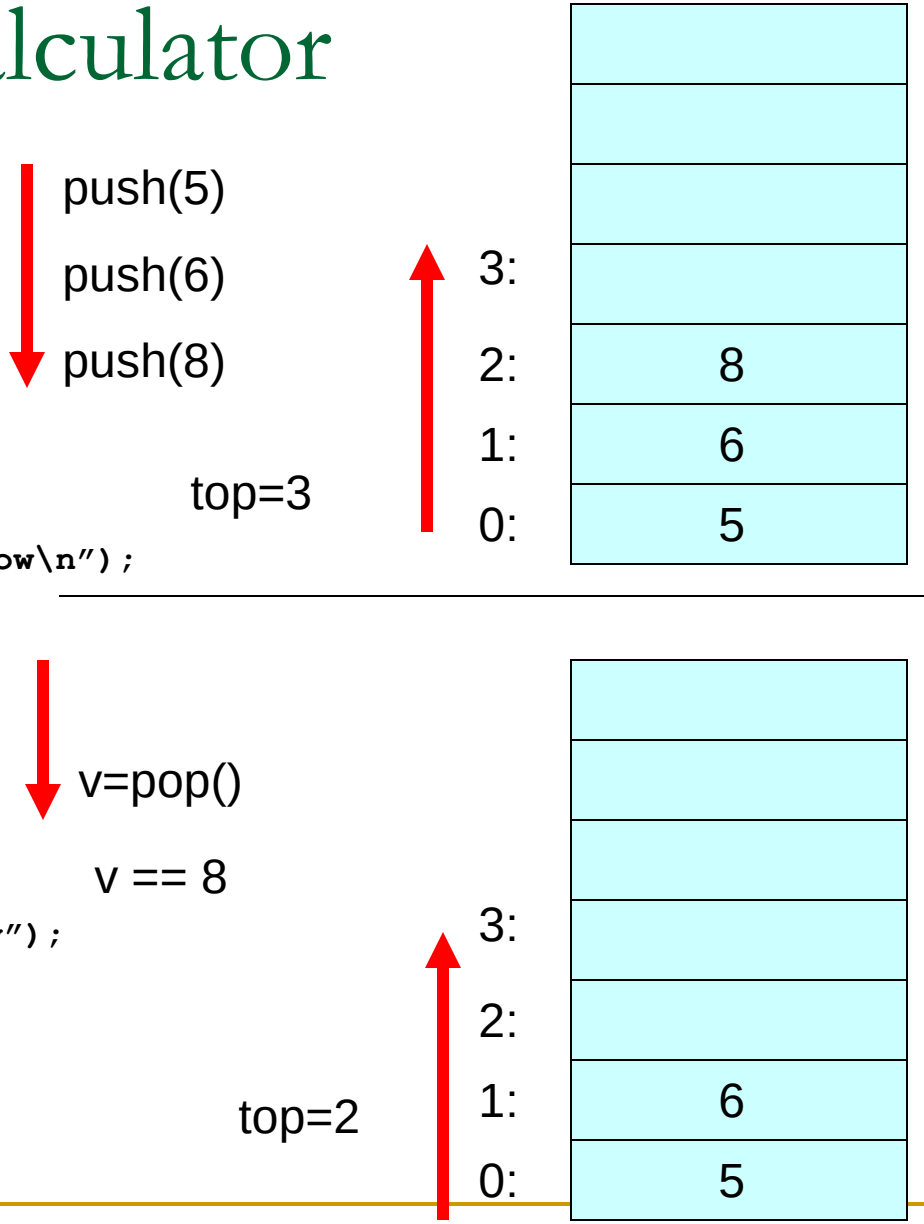
```
void a_byref(int *x){
        printf("x=%d\n",*x);
        *x = *x+1;
}
...
int y = 5;
a_byref(&y); // pass a pointer to y
printf("y = %d\n",y); // output: y = 6
```

Swap Two Numbers Using Passing Arguments by Reference

```
void swap(int *px, int *py)
{
        int temp;
        temp = *py;
        *py = *px;
        *px =  temp;
}

int main(){
        int x,y;
        x = 5;
        y = 9;
        printf("Before x=%d y=%d\n", x,y,);
        swap(&x,&y);
        printf("After x=%d y=%d\n",x,y);
}
```

Lab 4: RPN Calculator

```
// Implementation of a stack
#define MAXSTACK 100
double stack[MAXSTACK];
int top = 0;
void push(double val)
{
    if ( top == MAXSTACK) {
      printf("push: Stack overflow\n");
         exit(1);
    }
   stack[top] = val;
   top++;
}
double pop()
{
  if (top ==0) {
         printf("pop: Stack empty");
         exit(1);
  }
  top--;
  return stack[top];
}
```



Lab 4: RPN Calculator

```
int main( int argc, char ** argv) {
  // argv is the array of arguments
  int i;
  // Printing the arguments
  for (i=0; i < argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }

  // For all args
  //   If argv[i] is a number,
  //     convert it to double and push it to the stack.
  //   If argv[i] is "+"
  //     v1=pop(), v2=pop(), push(v1+v2)
  //   similar for other operands
  // end
  // result will be in stack[0]
}
```

Dangling Reference Problem

- The lifetime of local variables is limited to the time the function that contains the variables is called.
- You should not return a pointer to any of these variables.

Example:

```
int *m()
{
  int i;
  i=2;
  return &i;
}
main()
{
 int *x;
 x=m();
 printf("Hello");
 printf("*x=%d",*x);
}
```

OUTPUT is undefined *x=?????;

The space used by i in m() will be reused by other local variable in the first printf so the value of *x is undefined.

String Operations with Pointers

- We can rewrite many of the string functions using pointers

strlen using array operator

```
int strlen(char*s)
{
 int i=0;
 while(s[i])
 {
  i++;
 }
 return;
}
```

strlen using pointers

```
int strlen(char *s)
{

 int i=0;
 while(*s)
 {
  i++;
  s++;
 }

 return i;
}
```

strcpy using pointers

```
char *strcpy(char*dest, char *src)
{
 char*p =src;
 char*q=dest;

 while(*p)
 {
  *q=*p;
  q++;
  p++;
 }

 *q='\0';

 return dest;
}
```

Strcat using pointers

```
char *strcat(char*dest, char*src)
{
 char *p;
 // make p point to end of dest
 p=dest;
 while(*p) { p++; }
 q=src;
 //copy from *q to *p
 while(*q)
 {
  *p=*q;
  p++;
  q++;
 }
 *p='\0';
 return dest
}
```

Common Mistakes: Uninitialized pointers

- Uninitialized pointers

```
char*s;
strcpy(s,"Hello\n");
// wrong s is pointing to NULL or to an
// unknown value
// strcpy will cause SEGV because
// s is not pointing to valid memory
```

- Solution:

```
char s[7];
strcpy(s,"Hello\n");

or

char *s=(char*)malloc(20);
strcpy(s,"Hello\n");
```

Common Mistakes: Not enough space

- Not enough space. ***s*** needs to include ‘\0’ at the end of the string. “Hello\0” takes 6 chars and not 5.

  ```
  char s[5];
  strcpy(s,"Hello");
  ```

- Fix:

  ```
  char s[6];
  strcpy(s,"Hello");
  ```

Common Mistakes: Not enough space

```
char s[6];
strcpy(s,"Hello");
strcat(s,"world");
```

- s needs to have a tleast 12 characters to store "Hello world \0"
- The last strcat may give a SEGV or overwrite to the memory of other variables
- Solution: Allocate enough memory

```
char s[20];
strcpy(s,"Hello");
strcat(s," world");
```

strcmp(s1, s2) - String Comparison

```
r=strcmp(s1,s2);
r==0; //if s1 is equal to s2
r > 0 if s1 > s2
r < 0 if s1 < s2

strcmp("banana","apple"); > 0

apple
.
.
.
banana

strcmp("apple", "banana");< 0
```

strdup –duplicate string

- strdup creates a copy of the string using malloc

```
char*strdup(char *s1)
{
 char*s=(char *)malloc(strlen(s1)+1);
 strcpy(s,s1);
 return s;
}

 char a[20];
 strcpy( a,"Hello ");
 char*s=strdup(a);
```

- s and a are different strings with the same content.

Strings functions that check the string length

- The functions we have seen do not check for the length of destination string.
- There are equivalent functions that assume maximum of "n" characters in the destination. They are safer.

```
strncpy(char*dest, char *src, int n)
   //copies src in dest
   //a maximum of n chars
   // if src is longer than n chars
   // dest will not have '\0'
   // at the end otherwise it will put a
   //'\0' at the end of dest

strncat(char*dest, char *src, int n)
   //concatenates src into dest
   //upto n chars the length of
   //dest should be enough to
   //store dest and src. n includes both dest and src
```

Passing arrays as a parameter

- You can pass an array as a parameter by passing a pointer
- Arrays are pointers and viceversa

```
int sum(int a[], int size)
```

same as

```
int sum(int *a,int size)
```

Passing arrays as a parameter

```
int sum(int *a,int size)
{
	int sum=0;
	int i;

	for(i=0;i< size;i++)
	{
	 sum+=a[i];
	}
	return sum;
}

main()
{
	int a[]={7,8,3,2,1); // Initializing array

	n =sizeof(a)/sizeof(int);
	printf("sum(a)=%d\n",sum(a,n) );
}
```

Allocating arrays

- Allocating arrays statically

```
int array[200];
```

- Allocating arrays using malloc

```
int *array;
int n=200;
array=(int*)malloc(n*sizeof(int));
if(array==NULL)
{
  perror("malloc");
  exit(1);
}
```

realloc(oldblock, newsize)

- If you allocate memory with malloc then you can resize it.

  n=2*n;

  array=(int*)realloc(array, n*sizeof(int));

- realloc(oldblock, newsize) does the following:
  - Allocates a new block with the new size
  - Copies the old block into the new one block
  - Frees the old block
  - Returns a pointer to the new block

structs

- A struct is a compound type:

```
struct z
{
 int a;
 double x;
 char *s;
}r;
```

- If you want to refer to any of the members:

```
r.x = 23.5;
r.s = "Hello";
r.a = 3 ;
```

structs member variables

- You can name a struct so you can refer to it in multiple places

```
struct STUDENT
{
  char*name;
  double grade;
};
```

- Then define a variable as follows:

```
struct STUDENT peter, mary;
peter.name="peter";
peter.grade=100;
mary.name="mary";
mary.grade=100;
```

Pointers to struct

- If the struct variable is a pointer

```
struct STUDENT *p;
p=&peter;
```

- Then we can refer to fields of peter as follows:

```
(*p).name="peter";
(*p).grade=100;
```

- Since this way of referring to structs using pointers is common C has also the equivalent notation.

```
p->name="peter";
p->grade=100;
```

typedef struct

- Using typedef and structs we can also write:

```
typedef struct
{
 char*name;
 double grade;
}STUDENT;
STUDENT peter, mary;
```

- Or we can use both the struct name and typedef:

```
typedef struct STUDENT
{
 char*name;
 double grade;
}STUDENT;
```

- Now you can define variables as:

struct STUDENT peter, mary;
 //or
STUDENT peter, mary;

Single Linked List - Header File

single_linked_list.h:

```
typedef struct SLLENTRY
{
 char * name;
 char * address;
 struct SLLENTRY * next;//pointer to the next entry
};

typedef struct SLLIST{
   SLLENTRY * head;
} SLLIST;
```

Single Linked List - Header File

```
//Interface

// Initialize a new list
Void sllist_init(SLLIST * );

// Print List
void sllist_print(SLLIST * sllist);

// Given a name lookup the address. Return null if name not in list.
char * sllist_lookup(SLLIST * sllist, char * name);

// Add a new name, address to the list. Return 1 if name exist or 0ow
int sllist_add(SLLIST * sllist, char *name, char*address);

// Remove name from linked list. Return 1 if name exists or 0 otherwi
int sllist_remove(SLLIST*sllist, char*name);
```

Single Linked List - Test Main

sllist_test.c:

```
#include "single_linked_list.h"

int main() {
   SLLIST sl;
   int result;
   sllist_init(&sl);   // we create a single linked list sl


   //Add two items to the list
   result = sllist_add(&sl,"Peter Parker", "38 2nd,NY");
   result = sllist_add(&sl,"Clark Kent", "78 Super,Metro");

   // Print list
   sllist_print(&sl);
```

Single Linked List - Test Main

```
   char* addr  = sllist_lookup(&sl,"Clark Kent");
   if( addr == NULL) {
      printf("cannot find Clark's address\n");
      exit(1);
   }
   else
   {
      printf("Clark's address is %s\n", addr);
   }

   result = sllist_remove(&sl,"Clark Kent");
   if (result == 0)
   {
     printf("Cannot remove Clark's address\n");
     exit(1);
   }

   sllist_print(&sl);
}
```

Single Linked List - Initialize List

single_linked_list.c:

```
#include"single_linked_list.h"

// Initialize Linked list
void sllist_init(SLLIST * sl)
{
    // the list is initially empty so we initialize it to NULL
    //This is equivalent to saying (*sl).head = NULL
    sl->head = NULL;
}
```

Single Linked List - Print List

```
void sllist_print(SLLIST * sl)
{
  // Traverse the list and print each element
  SLLENTRY *p;
  p = sl->head;  //we initialize p to the head
  while(p != NULL) {
    printf("name = %s addr = %s\n",
           p->name, p->address);
    p = p->next;
  }
}
```

Single Linked List - Lookup

```
char sllist_lookup(SLLIST*sl, char* name)
{
   SLLENTRY *p;
   p = sl->head;
   while(p != NULL)
   {
        if(strcmp(name,p->name) == 0)
        {
                // YES ! we have found the name
           return(p -> address);
        }
        p = p->next;
   }
   // Now we are outside the while loop
   // this will happen if we have reached the end
   // of the list and still not found the name.
   // If this is the case then we return NULL

   return NULL;
}
```

Single Linked List – Add 1

```
int sllist_add(SLLIST* sl, char* name, char*address)
{
   SLLENTRY *p;
   // we first have to check if the name already exists
   p = sl->head // initialize p to head
   while(p != NULL)
   {
        if(strcmp(p->name,name) == 0)
        {
                // this means name already exists
                // if it already exists then we substitute
                // the old address for the new one
                // we need to free the previous address
                // since it was allocate with strdup
                free(p->address);
                p->address = strdup(address); //create a duplicate
                return(0);
        }
        p = p->next;  // incrementation for the while loop
   }
```

Single Linked List – Add 2

```
   // we have exited out of the while loop and
   // name does not exist. We need to create a new entry

   p = (SLLENTRY*)malloc(sizeof(SLLENTRY));

   // we make duplicate of name and address
   p->name = strdup(name);
   p->address = strdup(address);

   // we now put the new entry at the
   // beginning of the list

   p->next = sl->head;
   sl->head = p;

   return(1);
}
```

Single Linked List – Remove 1

```
int sllist_remove(SLLIST *sl, char*name);
{
    SLLENTRY* p = sl->head;
    SLLENTRY* prev = NULL;
    //prev is a pointer which points to the previous entry
    //we first have to find the entry to remove
    while(p != NULL)
   {
        if(!strcmp(p->name,name))
        {
              break; // entry is found
        }
        prev = p;
        p = p->next;
   }
```

Single Linked List – Remove 2

```
if(p == NULL)
{
     // element does not exist
     return 0;
}
// Now the entry pointed by p has the name we are looking for.
// prev points to the element before p

// There are two cases for p
// First Case: p is the first element. Therefore,
//    prev will point to NULL
// Second case: p is an internal node (includes the
//    last node in this case)
```

Single Linked List – Remove 3

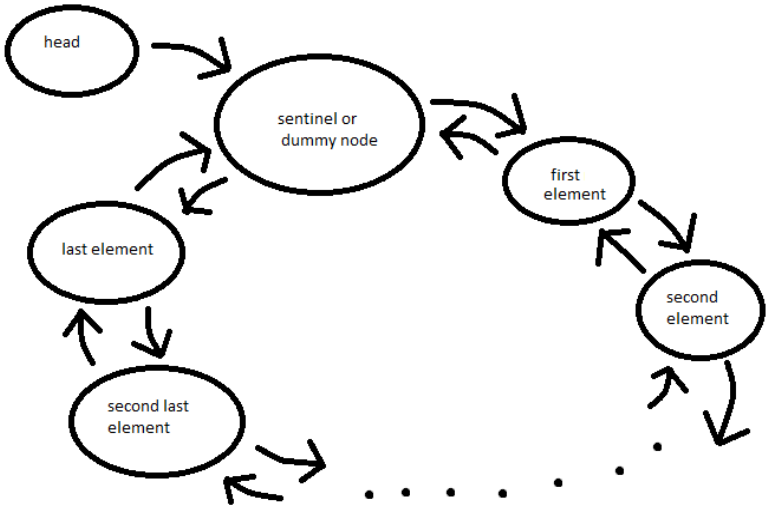
```
    if (prev == NULL) {
       sl->head = p->next;
    }
    else
    {    prev->next = p->next ;
    }
    // Now we have skipped the element but we are not
    // yet done removing it since we have to free the memory
    // Before we free p , we have to free the address and
    // the name associated with p because we used strdup
    // to allocate memory to it

    free(p->name);
    free(p->address);
    free(p);

    // the order here is critical, we first free the name and address since
       p points to them

    return 1;
}
```

Double Linked Lists

- Each node also has a previous that point sto the previous element of the list.
- They have faster access to insert or remove two elements from front or back of the list due to their bi-directional nature.
- They use an extra node called sentinel or dummy node to delimit the list
- The main advantage is that we can the beginning and the end of the list
- Double linked lists are more practical and versatile

Double Linked Lists

- By having a sentinel we will never need to modify the head except at the time of crating it.
- See DLList.h and DLList.c in lab3 for the implementation.



Pointer to Functions

- In the same way that you have pointers to data, you can have pointers to functions.
- Pointers to functions point to functions in the text segment.

Example:

```
//FUNCPTR is a type of pointer to a function
//that takes no arguments and returns void.
typedef void (* FUNCPTR)(void);

void hello(){
       printf("Hello world\n");
}

main(){
       FUNCPTR funcptr;
       Hello();

       //call hello through funcptr
       funcptr = hello;
      (* funcptr)();  //the same as calling funcptr();
}
Output:
   Hello world -> Printed by "hello()"
   Hello world -> Printed by (*funcptr)();
```

Use of pointer to functions: Sorting Any Array

- Polymorphism: You can write in C functions that can be used for variables of multiple types. Ex: Sorting function which is able to sort arrays of any type; Comparison function is passed as argument.

```
//generic pointer that can be used to point to any type
typedef int (*compare_func)(void *e1, void *e2);

// Function that compares two integers
int compInt(void *e1, void *e2)
{
        int * p1 = (int *)e1;
        int *p2 = (int *)e2;
        if(*p1 > *p2){ return 1; }
        else if(*p1< *p2){return -1;}
        else{return 0;}
}
```

Use of pointer to functions: Sorting Any Array

```
void sortAnyArray(void * array, int n, int elementsize,
                          compare_func comp)
{
   // Temporal memory used for swapping
   void * tmp = malloc(elementsize)
   int i, j;
   //use bubble sort
   for(i=0; i<n; i++)
   {
       for(j=0; j<i;j++)
       {
                //compute pointer to entry j
                void *e1 = (void*)((char*)array+j*elementsize);

                //The reason we convert to char* because
                //it is not possible to do pointer
                //arithmetic with void

                //compute pointer to entry j+1
                void *e2 = (void*)((char*)array+(j+1)*elementsize);
```

Use of pointer to functions: Sorting Any Array

```
            //sort in ascending order
            //swap if e1>e2
            //element at j is larger than in j+1
            if((*comp)(e1,e2)>0)
            {
                    //now we need to swap
                    //we need to swap the entries pointed by e1 and e2;
                    memcpy(tmp, e1,elementsize);
                    memcpy(e1,e2,elementsize);
                    memcpy(e2, tmp, elementsize;)
            }
        }
    }
    // Free memory used for swap
   free(tmp);
}
```

Use of pointer to functions: Sorting Any Array

```
//Using sorting function
int main(){
	// Sorting array of type int
	int a[] = {7,8,1,4,3,2}
	int n=sizeof(a)/sizeof(int);
	sortAnyArray(a, n, sizeof(int), compInt);
	int i=0;
	for(i=0;i<n;i++){
		printf("%d \n", a[i]);
	}

	// Sorting array of type string
	char * strings[] = {"pear", "banana", "apple", "strawberry"}
	n = sizeof(strings)/sizeof(char*);
	sortAnyArray(strings, n, sizeof(char*), compstr);
	for(i=0; i<n; i++)
	{
		printf("%s\n", strings[i]);
	}
}
```

Use of pointer to functions: Sorting Any Array

```
// String comparison
int compstr(void *e1, void *e2)
{
        char **p1 = (char**)e1;
        char **p2 = (char**)e2;
        //p1, p2 is a pointer to the string
        if((strcmp(*p1, *p2)>0) {
          return 1;
        }
        else if((strcmp(*p1, *p2)<0)
        {
          return -1;
        }
        else
        {
          return 0;
        }
}
```

Example of Pointers to Functions: Iterating over a List

- We can use pointers to functions to iterate over a data structure and call a function passed as parameter in every element of the data structure.
- This is called an iterator or mapper

Implementation of llist_mapper:

single_linked_list.h:

… **Other definitons**…

```
typedef void (*SLLISTFUNC)(char* name, char* value);
```

Example of Pointers to Functions: Iterating over a List

single_linked_list.c

```
//call llistfunc() in every element of linked list
void sllist_mapper(SLLIST *sl, SLLISTFUNC func))
{
      SLLENTRY *e;
      e = sl->head;
      while(e != NULL) {
        (*func)(e->name, e->value);
        e = e->next;
      }
}
```

Example of Pointers to Functions: Iterating over a List

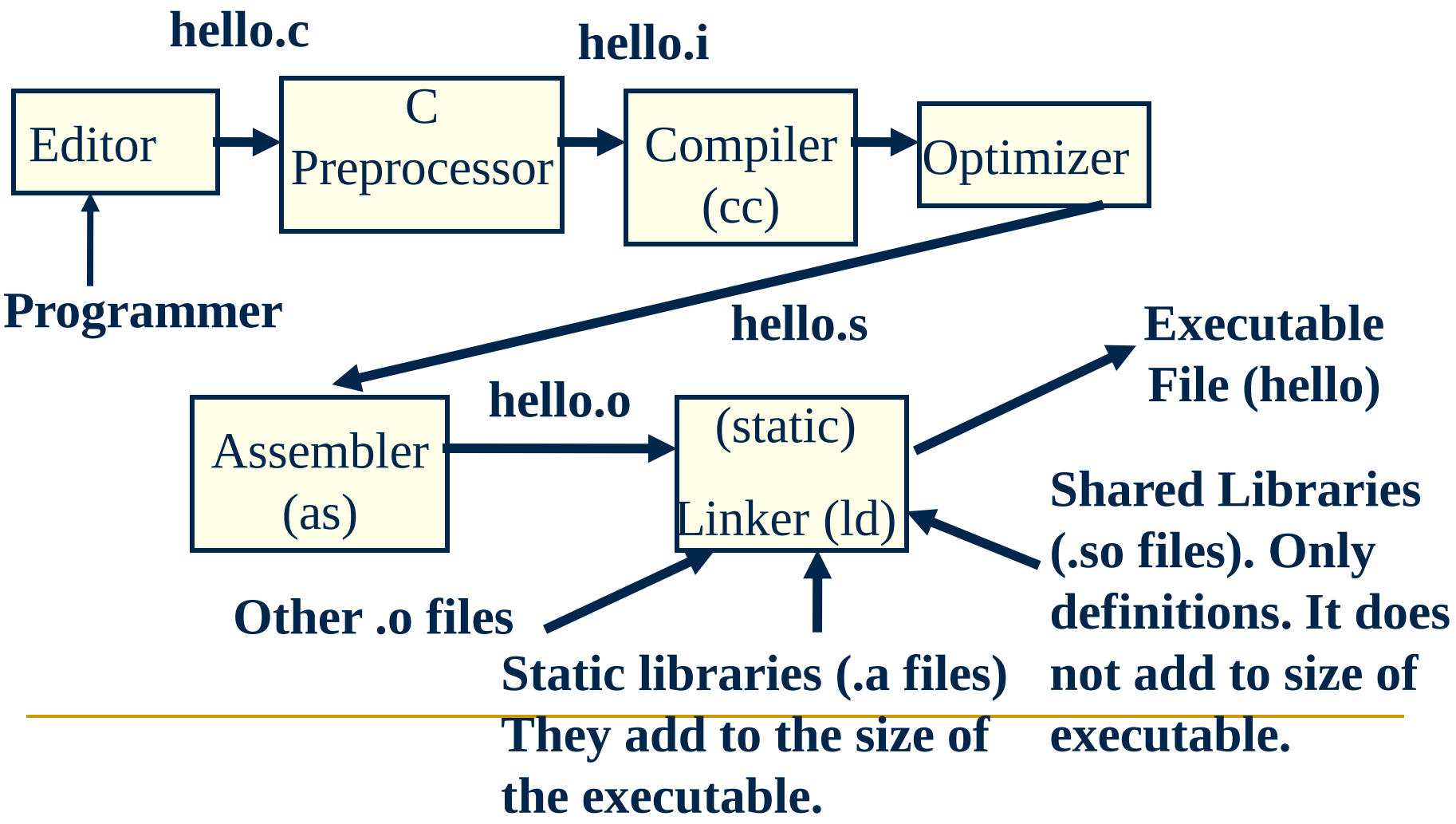
main.c

```
#include "linked_list.h"
void printEntry(char *name, char* value)
{
        printf("name:%s  value:%s\n", name, value);
}

Void printNamesWithA(char* name, char* value)
{
        if(name[0] =='A' || name[0] == 'a')
        {
          printf("name = %s value = %s\n", name, value);
        }
}
```

Example of Pointers to Functions: Iterating over a List

```
main()
{
        //read a linkedlist from disk
        SLLIST list;
        list = llist_init(&llist);
        list_read(&llist, "friends.rt");

        // Print all entries
        sllist_mapper(&list, printEntry);

        // print only entries that start with "a"
        list_mapper(&list, printNamesWithA);

}
```

Building a Program



Building a Program

- The programmer writes a program hello.c
- The ***preprocessor*** expands #define, #include, #ifdef etc preprocessor statements and generates a hello.i file.
- The ***compiler*** compiles hello.i, optimizes it and generates an assembly instruction listing hello.s
- The ***assembler*** (as) assembles hello.s and generates an object file hello.o
- The compiler (cc or gcc) by default hides all these intermediate steps. You can use compiler options to run each step independently.

Building a program

- The linker puts together all object files as well as the object files in static libraries.
- The linker also takes the definitions in shared libraries and verifies that the symbols (functions and variables) needed by the program are completely satisfied.
- If there is symbol that is not defined in either the executable or shared libraries, the linker will give an error.
- Static libraries (.a files) are added to the executable. shared libraries (.so files) are not added to the executable file.

Original file hello.c

```
#include <stdio.h>
main()
{
   printf("Hello\n");
}
```

After preprocessor

gcc -E hello.c > hello.i

(-E stops compiler after running preprocessor)

hello.i:

```
/* Expanded /usr/include/stdio.h */
 typedef void *__va_list;
 typedef struct __FILE  __FILE;
 typedef int     ssize_t;
 struct FILE {…};
 extern int fprintf(FILE *, const char *, ...);
 extern int fscanf(FILE *, const char *, ...);
 extern int printf(const char *, ...);
 /* and more */
 main()
 {
    printf("Hello\n");
 }
```

After assembler

gcc -S hello.c (-S stops compiler after assembling)

hello.s:

```
        .align 8
.LLC0:  .asciz  "Hello\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:   save    %sp, -112, %sp
        sethi   %hi(.LLC0), %o1
        or      %o1, %lo(.LLC0), %o0
        call    printf, 0
        nop
.LL2:   ret
        restore
```

After compiling

- "gcc -c hello.c" generates hello.o
- hello.o has undefined symbols, like the ***printf*** function call that we don't know where it is placed.
- The main function already has a value relative to the object file hello.o

csh> nm -xv hello.o

```
hello.o:
[Index]   Value      Size       Type  Bind  Other Shndx   Name
[1]     |0x00000000|0x00000000|FILE |LOCL |0     |ABS    |hello.c
[2]     |0x00000000|0x00000000|NOTY |LOCL |0     |2      |gcc2_compiled
[3]     |0x00000000|0x00000000|SECT |LOCL |0     |2      |
[4]     |0x00000000|0x00000000|SECT |LOCL |0     |3      |
[5]     |0x00000000|0x00000000|NOTY |GLOB |0     |UNDEF  |printf
[6]     |0x00000000|0x0000001c|FUNC |GLOB |0     |2      |main
```

After linking

- "**`gcc –o hello hello.c`**" generates the hello executable
- Printf does not have a value yet until the program is loaded

```
csh> nm hello
[Index]   Value      Size       Type  Bind  Other Shndx   Name
[29]    |0x00010000|0x00000000|OBJT |LOCL |0     |1      |_START_
[65]    |0x0001042c|0x00000074|FUNC |GLOB |0     |9      |_start
[43]    |0x00010564|0x00000000|FUNC |LOCL |0     |9      |fini_dummy
[60]    |0x000105c4|0x0000001c|FUNC |GLOB |0     |9      |main
[71]    |0x000206d8|0x00000000|FUNC |GLOB |0     |UNDEF  |atexit
[72]    |0x000206f0|0x00000000|FUNC |GLOB |0     |UNDEF  |_exit
[67]    |0x00020714|0x00000000|FUNC |GLOB |0     |UNDEF  |printf
```

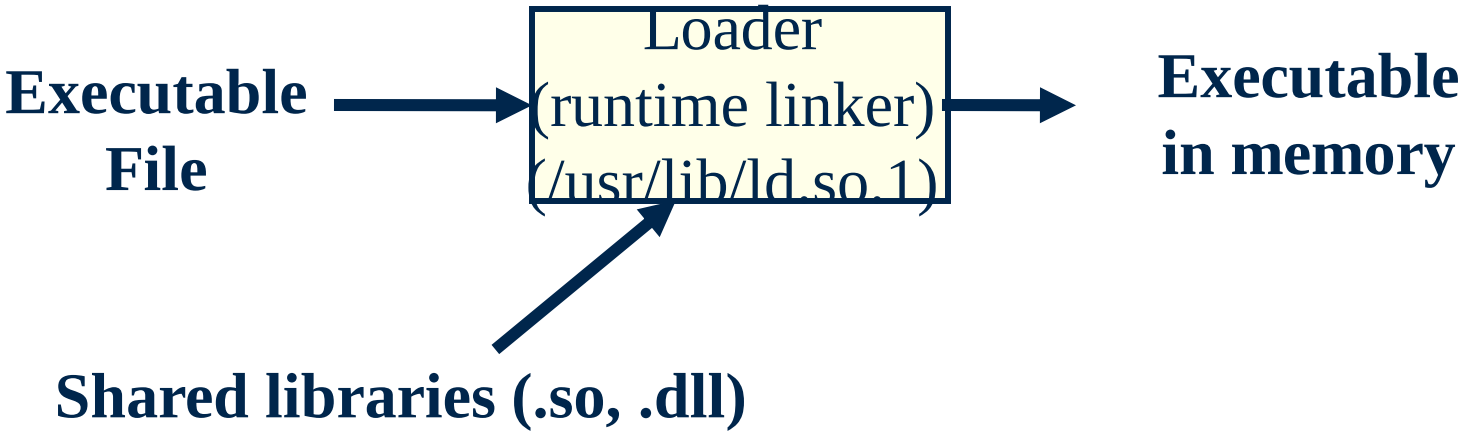
Loading a Program

- The loader is a program that is used to run an executable file in a process.
- Before the program starts running, the loader allocates space for all the sections of the executable file (text, data, bss etc)
- It loads into memory the executable and shared libraries (if not loaded yet)

Loading a Program

- It also writes (resolves) any values in the executable to point to the functions/variables in the shared libraries.(E.g. calls to printf in hello.c)
- Once memory image is ready, the loader jumps to the ***_start*** entry point that calls init() of all libraries and initializes static constructors. Then it calls ***main()*** and the program begins.
- ***_start*** also calls ***exit()*** when ***main()*** returns.
- The loader is also called “runtime linker”.

Loading a Program



Static and Shared Libraries

- Shared libraries are shared across different processes.
- There is only one instance of each shared library for the entire system.
- Static libraries are not shared.
- There is an instance of an static library for each process.

The C Preprocessor: Macro Definitions #define

- Macro Definitions are used to define constants or functions that need to be in-lined

```
#define PI 3.14
#define printHi printf("Hi")
```

- macros can have parameters

```
//returns true if c is lowercase…
 #define islower(c) (c>='a'&&c<='z')

 if(islower(x)){
     //Do something
 }
```

Be careful with macros

```
//returns true if c is lowercase…
   #define islower(c) (c>='a'&&c<='z')

     if(islower(x+1)){
         //Do something
     }
```

It becomes when expanded by the C preprocessor

```
    if( (x+1>='a'&& x +1<='z') ){
         //Do something
     }
```

However:

```
#define times2(x) x*x
y = times2(3+z)
Becomes
Y = 3+z*3+z // that is not what we want

To fix it use:
#define times2(x) ((x)*(x))
y = times2(3+z)
Becomes
Y = ((3+z)*(3+z)) // that is what we want
```

The C Preprocessor: Macro Definitions #define

- Another example of a macro

```
// get one character from a file
#define getchar() fgetc(stdin)
```

- A macro can be used from where it is defined to the end to the file

- Un-defining macros

  #undef PI

The C Preprocessor: File Inclusion #include

- Examples of file inclusion

```
#include “FILENAME”        //It will search for the file in the current directory----relative path
#include “/home/grr/a.h”     //Absolute path
#include <FILENAME>       //ook for include file in the system’s directories----/usr/include
```

- Example of an include file

mydefs.h:

```
#define PI 3.14
#define MAX_STUDENTS 14
```

myProgram.c

```
#include “mydefs.h”

int main(){
        Printf(“pi = %lf\n”, PI);
}
```

- We can even include a header file in another header file!

The C Preprocessor: Conditional Compilation #if

Examples of conditional compilation

#if constantexpr1 //evaluated by preprocessor, before compilation
… Included if constatnexpr is not 0
#end

#if constantexpr2
……
#else
……
#end

The C Preprocessor: Conditional Compilation #if

- Conditional Compilation is useful if we have various environment, say Solaris, Linux, Windows……

- Example

```
#define MACHINEARCH 64
     //Notice that the 64 here is subject to change from
    // one architecture to another
#if MACHINEARCH == 64
      //Code for 64 bits……
#elif MACHINEARCH ==32
     //Code for 32  bits
#else
      //Unknown
#end
```

The C Preprocessor: Conditional Compilation #if

- You may use conditional compilatiion to comment multiple lines of code that have comments already
- Example:

```
#if 0
   // We cannot use /**/ here to comment comments
   /*Hello*/
   int main(){
         /*Another comment*/
   }
#endif
```

The C Preprocessor: Conditional Compilation #ifdef and #ifndef

- #ifndef is often used in include files to prevent include files from being included multiple times.

stdio.h:

```
#ifndef STDIO_H
#define STDIO_H
                    //This code is included only once…
#endif
```

---

hello.h

```
#include <stdio.h>
```

---

hello.c

```
#include <stdio.h>   //include stdio.h above
#include "hello.h"   // hello.h will not include stdio.h again.

main()
```

Some Predefined Macros

Some predefined macros:

- _ _LINE_ _ Expands to the line number
- _ _FILE_ _ Expands to the file name
- _ _TIME_ _ Expands to the time of translation (compilation)

- EXAMPLE

```
#define MyAssert(x) \
    if(!(x)){ \
     printf("Assertion failed!%s:%d\n", \
           _ _FILE_ _, _ _LINE_ _); \
          }

main{
        …
            MyAssert(i>0 && i<MAX);
            a[i] = 5;
```

Assertions

- Assertions like the one above are useful for “defensive” programming.
- It is better to have an assertion failure than a segmentation fault.
- Assertions are already defined in

  #include <assert.h>

```
main{
 …
        assert(i>0 && i<max)
…
}
```

- If the expression iin the assertion is false, it will print will print the expression, file name, and line number

Bit Operations: Left and Right Shift << >>

- x >> i
  - Shifts bits of a number x to the right i positions
- x << i
  - Shifts bits of a number x to the left i positions
- Example:

```
int i, j;
i = 5; // In binary i is 00000101
j= 5 << 3; // In binary j is 00101000
printf("i=%d j=%d\n"); // Output: i=5 j=40
```

Left and Right Shift << >> with sign extension.

- When using i >> n (shift right) the behavior will change if i is a signed int or i is unsigned int.
- If i is signed int then i >> n will add *n* 1s in the left side.
- If i is unsigned then i >> n will add *n* 0s in the right side.
- This is called “signed extension”.

Example:

```
int i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011
int j = (i >> 2);  // j=11111111 11111111 11111111 11111110 =0xFFFFFFFFFE
```

and

```
unsigned i = 0xFFFFFFFB; // i=11111111 11111111 11111111 11111011
unsigned j = (i >> 2); // j=00111111 11111111 11111111 11111110=3FFFFFFE
```

Bitwise Operations: OR |

- The "|" operator executes "OR" bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y = (x | 0x2);
          // 00000101 | 00000010=00000111
printf("x=0x%x 0x%x\n", x,y); // x=0x5 y=0x7
```

Bitwise Operations: AND &

- The “&” operator executes “AND” bit operation.

```
unsigned x = 0x05; // 00000101
unsigned y =(x | 0x3);// 00000101 | 00000011 =00000001
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0x1
```

Bitwise Operations: XOR ^

- The “^” operator executes “XOR” bit operation.
- XOR : 0^0==0, 0 ^1 == 1, 1^0==1, 1^1==0

```
unsigned x = 0x05; // 00000101
unsigned y =(x ^ 0x3);// 00000101 ^ 00000011 =00000110
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0x6
```

Bitwise Operations: NOT ~

- The “~” negates bits.

```
unsigned x = 0x05; // 0000000 0000000 0000000 00000101
unsigned y = ~x;
    // ~00000101 = 11111111 11111111 11111111 11111010
printf(“x=0x%x 0x%x\n”, x,y); // x=0x5 y=0xFFFFFFFA
```

Using Bitwise Operations: Test if bit i is set:

```
int i = 4;
unsigned x = 23; // x = 00010111

// Test if bit i is set in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00010000

// Test if bit i is set
unsigned y = (x & mask); // y = 00010111 & 00010000 =00010000
int bit = (y >> i);      // bit = 00000001
                         // bit i in x is set.
```

Using Bitwise Operations: Set bit i :

```
int i = 3;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 0001000

// Set bit i
unsigned y = (x | mask); // y = 00010111 | 0001000 = 00011111
```

Using Bitwise Operations: Clear bit i :

```
int i = 2;
unsigned x = 23; // x = 00010111

// Set bit i in x
// Create mask with bit i set.
unsigned mask = (1 << i); // mask == 00000100
unsigned mask0 = ~mask;  // mask0 == 11111011

// Clear bit i
unsigned y = (x & mask0); // y = 00010111 & 11111011 = 00010011
```

Unions

- A union is like a struct but all the elements use the *same memory*.
- That means that modifying one element will overwrite the other elements.
- Unions are used to see the same memory area as different types.

- Example:

```
#include <stdio.h>
union A {
  int i;
  double f;
  char s[4];
};

main() {
        union A x;
        x.i = 65;
        printf("x.i=%d\n",x.i);
        printf("x.s=%s\n",x.s);
        printf("x.f=%lf\n",x.f);
}

grr@data ~/tmp $ ./a
x.i=65
x.s=A
x.f=0.000000
```